Λειτουργικά Συστήματα Εργαστηριακή Άσκηση

Ενδεικτικά:

Προαπαιτούμενο για την ομαλή κατανόηση του report είναι να δοθεί προσοχή στα σημεία που αναγράφεται «σύνοψη,γενικά,σημείωση» και τα υπογραμμισμένα πεδία. Να σημειωθεί ότι γίνεται πλήρης ανάλυση για κάθε κομμάτι κώδικα με σκοπό την ανάλυση ιδεών και τρόπου διεκπεραίωσης της Άσκησης.

Πριν ξεκινήσουμε να διευκρινιστεί ότι είναι ταυτόσημες στο report οι έννοιες:

1. Μέθοδος = λειτουργίες = benchmark = διαδικασία = πείραμα
2. εργασίες = διεργασίες = counts

Τέλος στο κομμάτι της kiwi.c και db.c περιγράφεται ο πλήρες και αναλυτικός τρόπος λειτουργία πολυνηματισμού. Μέχρι όμως να φτάσουμε σε αυτές τις ενότητες εξηγούμε μεταβλητές και εισαγωγικές συναρτήσεις ώστε να προετοιμάσουμε την διαδικασία περιγραφής.

Α) ΓΡΑΜΜΗ ΕΝΤΟΛΩΝ ΚΑΙ ΔΥΝΑΤΕΣ ΛΕΙΤΟΥΡΓΙΕΣ

i)Γραμμή εντολών:

- ./kiwi-source <read or write> <counts>
- ./kiwi-source <read or write> <counts> <threads> <random>
- ./kiwi-source <readwrite> <counts> <threads> <percentage> <random>
- ./kiwi-source <readwrite> <counts> <threads> <random>

Η εντολή kiwi-source είναι ένα βοηθητικό πρόγραμμα που μπορεί να χρησιμοποιηθεί για τη δημιουργία λειτουργιών ανάγνωσης ή εγγραφής σε ένα δέντρο LSM. Η εντολή διαθέτει αρκετές διαφορετικές επιλογές που μπορούν να χρησιμοποιηθούν για τον έλεγχο της συμπεριφοράς των διεργασιών που παράγει.

```
myy601@myy601lab1:~$ cd kiwi
myy601@myy601lab1:~/kiwi$ cd kiwi-source/
myy601@myy601lab1:~/kiwi/kiwi-source$ ./kiwi-bench readwrite 10000 40 80 1
```

ii)Ακολουθεί μια επεξήγηση των διαφόρων επιλογών της εντολής:

- **read ή write ή readwrite:** Αυτή η επιλογή καθορίζει αν οι λειτουργίες που θα παράγει η εντολή θα είναι λειτουργίες ανάγνωσης ή εγγραφής ή και τον δυο ταυτόχρονα.
- **counts:** Αυτή η επιλογή καθορίζει τον συνολικό αριθμό των διεργασιών που θα δημιουργηθούν από την εντολή.
- **threads:** Αυτή η επιλογή καθορίζει τον αριθμό των νημάτων που θα χρησιμοποιηθούν.
- **percentage:** Αυτή η επιλογή χρησιμοποιείται μόνο κατά τη δημιουργία λειτουργιών εγγραφής και ανάγνωσης (readwrite). Καθορίζει το ποσοστό των διεργασιών που θα είναι εγγραφές.
- **random:** Αυτή η επιλογή καθορίζει εάν τα κλειδιά που θα γράφει ή θα διαβάζει θα είναι τυχαία.

iii)Ακολουθούν ορισμένα παραδείγματα για το πώς μπορεί να χρησιμοποιηθεί η εντολή :

- **./kiwi-source read 10000**: Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης χωρίς τυχαία κλειδιά και 1 νήμα default.
- **./kiwi-source write 10000:** Αυτό θα δημιουργήσει 10.000 διεργασίες εγγραφής χωρίς τυχαία κλειδιά και 1 νήμα default.
- **./kiwi-source read 10000 4:** Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης, χρησιμοποιώντας 4 νήματα.
- **./kiwi-source read 10000 4 1:** Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης, χρησιμοποιώντας 4 νήματα και θα κάνει ανάγνωση τυχαία κλειδιά.
- **./kiwi-source readwrite 10000:** Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης-εγγραφή, χρησιμοποιώντας 2 νήματα που το καθένα θα έχει ίσα μοιρασμένες διεργασίες, όπου κάθε νήμα θα εκτελεί read και write. **(Θα εξηγηθεί παρακάτω).**
- **./kiwi-source readwrite 10000 40:** Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης-εγγραφής, χρησιμοποιώντας 40 νήματα, τα οποία αφού δεν υπάρχει ποσοστό θα μοιραστούν ισόποσα για την λειτουργία write και read, όπως και το ίδιο θα εφαρμοστεί και με τα counts(θα εξηγηθεί στην kiwi.c παρακάτω).
- **./kiwi-source readwrite 10000 4 80:** Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης-εγγραφής, χρησιμοποιώντας 4 νήματα, και το 80% των διεργασιών θα είναι για εγγραφή και οι υπόλοιπες για ανάγνωση , όπως και το 80% των νημάτων θα είναι για εγγραφή και το 20% για ανάγνωση (εξήγηση στην kiwi.c).

- **./kiwi-source readwrite 10000 4 80 1:** Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης-εγγραφής, χρησιμοποιώντας 4 νήματα, και το 80% των διεργασιών θα είναι για εγγραφή και οι υπόλοιπες για ανάγνωση,όπως και με τα threads, και τα κλειδιά θα είναι τυχαία.

A)Γενικά αν θέλω τυχαία κλειδιά για read or write χωρίς πολυνηματική διαδικασία μπορώ να γράψω:
./kiwi-source read 10000 1 1: Αυτό θα δημιουργήσει 10.000 διεργασίες ανάγνωσης χωρίς τυχαία κλειδιά. Δηλαδή να δώσω 1 νήμα

B) Η readwrite λειτουργία έχει πάντα 2 νηματα ως default επιλογή. Και για να βάλω τυχαία κλειδιά πρέπει ο χρήστης να δώσει όλα τα ορίσματα μαζί με το ποσοστό και μετά να ορίσει μια θετική τιμή για το r. (βλέπε το last bullet από πάνω στο νούμερο (iii)).

C) Ο χωρισμός διεργασιών αναλύεται παρακάτω στην Kiwi.c όπως και των threads. (Η γενική ιδέα είναι ότι έχουμε πετύχει ομοιόμορφο διαμοιρασμό threads, count)

D) Αν ο χρήστης δώσει 1 thread στην readwrite τότε θα εκτελεστεί μόνο η λειτουργία του read με τα μισά counts που θα δώσει. Μπορούσαμε να το αλλάξουμε ώστε όταν ο χρήστης δώσει μόνο ένα νήμα να του εμφανίζουμε ότι χρειάζεται ελάχιστα 2.

E) Δεν γινεται τα threads kai counts να είναι μικρότερα η ίσα του μηδενός, και το ποσοστό μεγαλυτερο του 100 ή μικρότερο του 0.

Όλα αυτά θα αναλυθούν και λίγο παρακάτω.

Β) ΑΡΧΕΙΑ ΚΩΔΙΚΑ ΚΑΙ ΕΠΕΞΗΓΗΣΗ

«BENCH.C»:

a) "main()"

Λαμβάνει ορίσματα γραμμής εντολών για να καθορίσει τον τύπο του benchmark, τον αριθμό των λειτουργιών, τον αριθμό των νημάτων, το ποσοστό των λειτουργιών εγγραφής και το αν θα χρησιμοποιηθούν τυχαία κλειδιά. Ακολουθεί μια ανάλυση του κώδικα:

```
int main(int argc, char** argv) {
    if (argc < 3 || argc > 6) {
        fprintf(stderr, "Usage: db-bench <write|read|readwrite> <count> [<threads> <percentage> <random>]\n");
        exit(1);
    }
```

1. Ελέγχει αν ο αριθμός των ορισμάτων γραμμής εντολών είναι έγκυρος (μεταξύ 3 και 6).
    a. Αν όχι, εκτυπώνει οδηγίες σωστής χρήσης ορισμάτων.

```
strcpy(choice, argv[1]); //choice option (read,write,readwrite)
long int count =  atol(argv[2]); //total counts
long int threads = argc >= 4 ? atol(argv[3]) : 1;  //total threads
double percentage = argc >= 5 ? atof(argv[4]): 30000; //percentage for write , default value 30000
int r = argc >= 5 ? 1 : 0; //random keys-value for read or write
double total_cost = get_time(); //total benchmark time
```

2. Ανάλυση των ορισμάτων της γραμμής εντολών:
    a. **Choice**: Ο τύπος του benchmark (read,write, readwrite).
    b. **Count**: Ο συνολικός αριθμός των διεργασιών που πρέπει να εκτελεστούν.
    c. Threads: Ο αριθμός των νημάτων που θα χρησιμοποιηθούν
        i. **Οι προεπιλεγμένες τιμές** αν δεν οριστούν από τον χρήστη είναι:

            A)1 thread για read or write και

            B)2 threads για readwrite.

    d. Percentage: Το ποσοστό των λειτουργιών εγγραφής για το benchmark ανάγνωσης-εγγραφής
        i. **Οι προεπιλεγμένη τιμή** αν δεν οριστεί από τον χρήστη είναι: 30000, ο λόγος είναι για τον διαχωρισμό των readwrite σε παρακάτω συνθήκη που θα αναλυθεί περεταίρω.
    e. **r**: Εάν θα χρησιμοποιηθούν τυχαία κλειδιά για λειτουργίες ανάγνωσης ή εγγραφής (για readwrite εξηγούμε παρακάτω σε μια If condition):
        i. η τιμή 1 εάν argc >= 5(δηλαδή αν έχουμε τελευταίο όρισμα κάποιον αριθμό,
        ii. διαφορετικά ίσο με 0.
    f. Total_cost: Ξεκινάει ένας μετρητής χρόνου για ολόκληρο το benchmark, μέχρι την στιγμή που θα το σταματήσουμε στο τέλος της main, αφού θα έχει ολοκληρωθεί μέχρι κι το κλείσιμο της βάσης. (η **get_time()** θα αναλυθεί στην **kiwi.c**)

```
if (count <= 0 || threads <= 0 ||percentage<0 || (percentage> 100 && percentage != 30000)) {
        perror("Error Please try again but check the following reasons.\na)Negative or zero number of counts\nb)N
        exit(1);
    }
```

3. **Επικυρώνει τα ορίσματα count** και **threads** και **percentage**:
    a. Εάν είτε το count είτε τα threads είτε το ποσοστό είναι αρνητικά, αλλά και αν το ποσοστό είναι μεγαλύτερο του 100 εκτυπώνει ένα μήνυμα σφάλματος και τερματίζει.

Παράδειγμα εκτύπωσης λάθους:

```
Error Please try again but check the following reasons.
a)Negative or zero number of counts
b)Negative or zero number of threads
c)Percentage bigger than 100 or smaller or equal to 0.
```

```
_print_header(count); //different headers about the size
_print_environment(); //different enviroment variables about system
_open_db(); //open the database
```

4. **Ανοίγει την βάση δεδομένων** και **εκτυπώνει** πληροφορίες για κεφαλίδες και περιβάλλον που προϋπήρχαν στο πρόγραμμα.

```
//choose benchmark mode
if (strcmp(choice, "write") == 0 && argc <= 5) {
    _create_threads_test(count, threads, r, percentage); //create thread system benchmark or multithreading system benchmark
}else if (strcmp(choice, "read") == 0  && argc <= 5) {
    _create_threads_test(count, threads, r, percentage); //create thread system benchmark or multithreading system benchmark
}else if (strcmp(choice, "readwrite") == 0) {
    if (argc == 3){
        threads = 2; //default value for threads in readwrite benchmark if the user doesn't give us threads
        _create_threads_test(count, threads, r, percentage); //create multitheading system benchmark for readwrite
    }else{
        if(percentage > 0 && argc >= 6 ) //if we have 6 arguements in readwrite it means random key values
            r = 1;
        else
            r = 0; //else don;t use random keys
        _create_threads_test(count, threads, r, percentage); //create the multithrading system benchmark for readwrite
    }
 }else{
    fprintf(stderr, "Invalid test type.\n");
    exit(1);
    }
```

5. **Εκτέλεση benchmark** με βάση τη μεταβλητή **choice**:

a) Εάν η **επιλογή είναι "write"** και argc <= 5(δηλαδή μέχρι 5 ορίσματα που παίρνει η write), καλεί την συνάρτηση create_threads_test() η οποία δημιουργεί τα νήματα με βάση τα ορίσματα που αναφέραμε προηγουμένως (θα αναλυθεί παρακάτω).
b) Εάν η **επιλογή είναι "read"** και argc <= 5, τότε καλείται αντίστοιχα η ίδια συνάρτηση όπως στην write.
c) Εάν η **επιλογή είναι "readwrite"**:
    i) **Εάν argc == 3**, ορίζει τον προεπιλεγμένο αριθμό νημάτων σε 2 όπως αναφέραμε προηγουμένως και εκτελεί ένα

benchmark ανάγνωσης-γραφής με την συνάρτηση που προαναφέραμε.

ii) **Αν percentage > 0 και argc >= 6**, χρησιμοποιεί **<u>τυχαία κλειδιά</u>** για το benchmark ανάγνωσης-γραφής.

iii) **Διαφορετικά, <u>δεν χρησιμοποιεί τυχαία κλειδιά</u>** για το benchmark readwrite και <u>κρατάει το ποσοστό που δίνει ο χρήστης ή το default</u> αν αυτό δεν οριστεί.

d) Εάν **η επιλογή δεν είναι κανένα από τα παραπάνω**, εκτυπώνει ένα μήνυμα σφάλματος και τερματίζει.

6. Λίγο πριν τον τερματισμό:

```
_close_db(); //close db
total_cost = get_time()-total_cost; //stop the benchmark clock
//print the resluts after the multithreading benchmark
if (percentage == 30000) //if default percentage that means, the user doesn't give any percentage for the bench
    percentage = 0; // so update the value to 0 for the print porpuses
prints_for_results(sum_of_cost[0],sum_of_cost[1],threads,count,percentage,total_cost);

return 1;
```

a) Κλείνει την βάση δεδομένων.

b) Σταματάει τον μετρητή χρόνου για το benchmark, τον οποίο αρχικοποιήσαμε πιο πάνω στην αρχή του bench.

c) <u>Εκτυπώνει τα στατιστικά αποτελέσματα</u> των benchmark (που θα αναφερθούν αμέσως μετά) και τον <u>συνολικό χρόνο εκτέλεσης με πολλαπλά νήματα.</u>

d) Πριν εκτυπώσει τα στατιστικά , <u>αν δεν δοθεί από τον χρήστη κάποιο ποσοστό</u> (για readwrite) ή εκτελέσει σκέτο benchmark για read ή write, τότε καθαρά <u>για την εκτύπωση στην οθόνη</u> και μόνο, <u>το θέτουμε σε 0.</u>

b) “print_results()”

```
int prints_for_results(long double total_time_write,long double total_time_read, long int threads,long int counts,double percentage,double total_cost)
{
```

Η συνάρτηση εκτυπώνει στατιστικά στοιχεία σχετικά με την απόδοση του συστήματος με βάση τον αριθμό των λειτουργιών και τα threads και τον συνολικό χρόνο που απαιτείται για κάθε τύπο λειτουργίας. Λαμβάνει 6 παραμέτρους: **total_time_write, total_time_read ,threads,counts,percentage,total_cost.**

<u>Ακολουθεί μια ανάλυση του κώδικα:</u>

1) Εκτύπωση γενικών στατιστικών του benchmark:

```
printf(LINE);
printf("\n-------------STATISTICS----------------\n");
printf(" Benchmark mode -> %s \n Total threads -> %ld\n Total Counts -> %ld\n Percentage -> %.2f/100.00\n Total bench cost -> %f\n",
    choice,threads,counts,percentage,total_cost);
```

   a. της μεθόδου benchmark που επιλέχτηκε η οποία είναι αποθηκευμένη στην μεταβλητή "choice"
   b. τον αριθμό των συνολικών νημάτων που χρησιμοποιήθηκαν στο benchmark. (threads)
   c. Τον αριθμό των διεργασιών που εκπληρώθηκαν (counts)
   d. Το ποσοστό εγγραφών που όρισε ο χρήστης (percentage)
   e. Το συνολικό χρόνο εκτέλεσης του benchmark (total_cost)

   Παράδειγμα εκτύπωσης:

```
-------------STATISTICS-----------------
 Benchmark mode -> readwrite
 Total threads -> 50
 Total Counts -> 100000
 Percentage -> 33.00/100.00
 Total bench cost -> 0.732741
+------------------------------+----------
```

2) Η συνάρτηση ελέγχει την τιμή της μεταβλητής choice για να καθορίσει για ποιο τύπο λειτουργίας θα εκτυπώσει στατιστικά στοιχεία:

```
//print statistics for write
if (strcmp(choice, "write")==0)
{
    printf(LINE);
    printf("------------- RESULTS -------------\n");
    printf("a)Writes -> done:%ld\nb)Seconds/operation -> %.6Lf sec/op \nc)Writes/second(estimated) -> %.1Lf writes/sec \nd)
        sum_of_counts[0],
        total_time_write / sum_of_counts[0],
        sum_of_counts[0] / total_time_write,
        threads_for_rw[0],
        total_time_write / threads_for_rw[0],
        total_time_write);
}
```

   a. Εάν η **<u>επιλογή είναι "write":</u>** εκτυπώνει στατιστικά στοιχεία για τη λειτουργία εγγραφής:
      i. τον αριθμό των διεργασιών που πραγματοποιήθηκαν,
      ii. τον μέσο χρόνο ανά λειτουργία εγγραφής,
      iii. τον εκτιμώμενο αριθμό εγγραφών ανά δευτερόλεπτο,
      iv. τα threads που χρησιμοποίησε,
      v. τον εκτιμώμενο μέσο χρόνο που έκανε ανά thread,
      vi. το συνολικό κόστος σε δευτερόλεπτα.

```
else if (strcmp(choice, "read")==0)
{
    printf(LINE);
    printf("------------ RESULTS ------------\n");
    printf("a)Reads -> done:%ld\nb)Found -> %ld\nc)Seconds/operation -> %.6Lf sec/op \nd)Read
        sum_of_counts[1],found_keys,
        total_time_read / sum_of_counts[1],
        sum_of_counts[1] / total_time_read,
        threads_for_rw[1],
        total_time_read / threads_for_rw[1],
        total_time_read);
```

b. Εάν η επιλογή είναι "read": εκτυπώνει στατιστικά στοιχεία για την λειτουργία ανάγνωσης:

- **i.** τον αριθμό των αναγνώσεων που πραγματοποιήθηκαν,
- **ii.** τον αριθμό των κλειδιών που βρέθηκαν,
- **iii.** τον μέσο χρόνο ανά λειτουργία ανάγνωσης,
- **iv.** τον εκτιμώμενο αριθμό αναγνώσεων ανά δευτερόλεπτο,
- **v.** τα threads που χρησιμοποίησε,
- **vi.** τον εκτιμώμενο μέσο χρόνο που έκανε ανά thread,
- **vii.** το συνολικό κόστους σε δευτερόλεπτα.

```
        total_time_read);
//print statistics for readwrite
}else if (strcmp(choice, "readwrite")==0){
    printf(LINE);
    printf("------------ WRITE RESULTS ------------\n");
    printf("a)Writes -> done:%ld\nb)Seconds/operation -> %.6Lf sec/op \nc)Writes/second(e
        sum_of_counts[0],
        total_time_write / sum_of_counts[0],
        sum_of_counts[0] / total_time_write,
        threads_for_rw[0],
        total_time_write / threads_for_rw[0],
        total_time_write);
    printf(LINE);
    //printf(LINE);
    printf("------------ READ RESULTS ------------\n");
    printf("a)Reads -> done:%ld\nb)Found -> %ld\nc)Seconds/operation -> %.6Lf sec/op \nd)
        sum_of_counts[1],found_keys,
        total_time_read / sum_of_counts[1],
        sum_of_counts[1] / total_time_read,
        threads_for_rw[1],
        total_time_read / threads_for_rw[1],
        total_time_read);
```

c. Εάν η επιλογή είναι "readwrite": εκτυπώνει στατιστικά στοιχεία τόσο για τις λειτουργίες ανάγνωσης όσο και για τις λειτουργίες εγγραφής:

- **i.** συμπεριλαμβανομένων των ίδιων πληροφοριών όπως περιγράφθηκαν παραπάνω για την read και την write.

Παράδειγμα εκτύπωσης:

```
----+
------------- WRITE RESULTS -------------
a)Writes -> done:33000
b)Seconds/operation -> 0.000016 sec/op
c)Writes/second(estimated) -> 62059.0 writes/sec
d)Threads -> 16
e)Seconds/Thread(estimated) -> 0.033235 sec/th
f)Total cost -> 0.531752 sec
 +-----------------------------+---------------+
-----+
------------- READ RESULTS -------------
a)Reads -> done:67000
b)Found -> 67000
c)Seconds/operation -> 0.000007 sec/op
d)Reads/second(estimated) -> 135196.5 read/sec
e)Threads -> 34
f)Seconds/Thread(estimated) -> 0.014576 sec/th
g)Total cost -> 0.495575 sec
```

ΣΥΝΟΨΗ: Έχουμε **σωστή ενημέρωση μετρήσεων και απόδοσης** τα οποία εκτυπώνουμε στην οθόνη. Η εγκυρότητα των μετρήσεων θα αποδειχθεί και στα παρακάτω στην “kiwi.c”

<BENCH.H>:

Αυτό το header αρχείο περιλαμβάνει **μία δομή για τα threads, πρωτότυπα συναρτήσεων και global μεταβλητές.**

```c
//struct to keep the counts,r and a flag for the type of operation for every thread.
typedef struct{
    long int count; //counts for the thread
    int r;  //random keys
    int read_or_write; //if it is read(0) or write(1) thread
    int timeflag; //time flag
}threads_data;
```

a. **Struct threads_data:**
   i. count: Ο αριθμός των διεργασιών που ανατίθενται σε ένα νήμα.
   ii. r: Μια σημαία που υποδεικνύει αν θα χρησιμοποιηθούν τυχαία κλειδιά για τις λειτουργίες benchmark
   iii. read_or_write: Μια σημαία που υποδεικνύει τον τύπο λειτουργίας για ένα νήμα (0 για ανάγνωση, 1 για εγγραφή).

iv. timeflag: Μια σημαία που σχετίζεται με τη μέτρηση του χρόνου. Με την τιμή 1 υποδεικνύουμε ότι είναι το τελευταίο νήμα ανάγνωσης ή εγγραφής για να σταματήσει ο χρόνος καταγραφής λειτουργίας. (θα εξηγηθεί και παρακάτω).

b. **get_time():** Πρωτότυπο συνάρτησης για τη λήψη της τρέχουσας ώρας σε δευτερόλεπτα, η οποία θα αναλυθεί παρακάτω στην χρονομέτρηση.

```
double get_time(); //get time in sec
void _create_threads_test(long int count,long int threads,int r,double percentage); //for the multithreading creation system
```

c. **_create_threads_test():** Πρωτότυπο συνάρτησης για τη δημιουργία και τη διαχείριση νημάτων για το benchmark. Λαμβάνει τα ορίσματα που αναλύσαμε προηγουμένως στην γραμμή εντολών.

```
DB* db; //pointer for db.
char choice[10]; //to choose operation
long int found_keys; // how many keys have been found from read
long int sum_of_counts[2];// usefull for the statistics for the counts of write[0] and read[1]
long double sum_of_cost[2]; //statistic for costs ( write[0] and read[1] )
long int threads_for_rw[2];//threads for read and for write
```

d. Global μεταβλητές: (Είναι οι μεταβλητές που χρησιμοποιούνται και είναι χρήσιμες κυρίως για συνάρτηση εκτύπωσης στατιστικών **print_for_results()** που προαναφέραμε)

i. **Choice**: Ένας πίνακας χαρακτήρων για την αποθήκευση του τύπου benchmark που αναφέραμε στην main.

ii. **Found_keys**: Ένας μετρητής για τον αριθμό των κλειδιών που βρέθηκαν κατά τη διάρκεια των νημάτων ανάγνωσης.

iii. **sum_of_counts:** Ένας πίνακας για την αποθήκευση των μετρήσεων των λειτουργιών εγγραφής (δείκτης 0) και ανάγνωσης (δείκτης 1).

iv. **sum_of_cost:** Ένας πίνακας για την αποθήκευση του κόστους (χρόνου) της εγγραφής (δείκτης 0) και ανάγνωσης (δείκτης 1).

v. **DB* db:** Μια μεταβλητή για τον δείκτη της βάσης δεδομένων.

vi. **threads_for_rw:** Ένας πίνακας για την αποθήκευση των threads που χρησιμοποίησε κάθε μέθοδος read(1) or/and write(0)

Σημείωση: **Το σημαντικό είναι ότι ανοίγουμε την βάση στην αρχή και την κλείνουμε στο τέλος του προγράμματος**. Αυτό θα αναφερθεί και στην συνέχεια του προγράμματος αφού μας είναι χρήσιμο στην πολυνηματική λειτουργία με **ταυτοχρονισμό**.

«KIWI.C»:

Σε αυτό το αρχείο θα αναλυθούν οι πρωτότυπες συναρτήσεις αλλά και οι τροποποιήσεις και νέες συναρτήσεις που έχουν προστεθεί με σκοπό την επίτευξη της πολυνηματικής διαδικασίας η οποία θα αναλυθεί στο τέλος αναλυτικά.

```
double get_time() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (double) tv.tv_sec + (double) tv.tv_usec / 1000000.0;
}
```

a. **get_time():** Αυτή η συνάρτηση επιστρέφει την τρέχουσα ώρα ως double τιμή. Χρησιμοποιεί τη συνάρτηση gettimeofday() για να λάβει την τρέχουσα ώρα και να τη μετατρέψει σε δευτερόλεπτα με ακρίβεια μικροδευτερολέπτων.

Σημείωση: **Δημιουργήσαμε δική μας συνάρτηση για την καταγραφή χρόνου για μεγαλύτερη ακρίβεια, αντικαθιστώντας τον δικό σας τρόπο καταγραφής χρόνου και για καλύτερη διευκόλυνση στο κάλεσμα.**

```
//to open db
void _open_db(){
    db = db_open(DATAS);
}
//to close db
void _close_db(){
    db_close(db);
}
```

c. _open_db(): Αυτή η συνάρτηση ανοίγει τη βάση δεδομένων χρησιμοποιώντας τη συνάρτηση db_open() και αναθέτει το αποτέλεσμα στην μεταβλητή db. **Αυτή η συνάρτηση καλείται όπως αναφέραμε στην "initialize()" στην αρχή του benchmark.**

d. _close_db(): Αυτή η συνάρτηση κλείνει τη βάση δεδομένων χρησιμοποιώντας τη συνάρτηση db_close() και την μεταβλητή db. **Αυτή η συνάρτηση καλείται στο τέλος του benchmark όπως δείξαμε στην "main()"**

```c
void* request_thread(void* args) {
    threads_data* t_args = (threads_data*) args;
    if(t_args->read_or_write == 1){
        _write_test(t_args->count, t_args->r,t_args->timeflag);
        pthread_exit(NULL);
    }else if( t_args->read_or_write == 0){
        _read_test(t_args->count, t_args->r,t_args->timeflag);
        pthread_exit(NULL);
    }
    return NULL;
}
```

e. request_thread(): Η συνάρτηση request_thread(), χρησιμοποιείται ως συνάρτηση νήματος για το benchmark. Η συνάρτηση δέχεται έναν δείκτη void* ως όρισμα, ο οποίος στη συνέχεια μετατρέπεται σε έναν δείκτη τύπου threads_data*. Ακολουθεί μια ανάλυση της συνάρτησης:
   i. Η συνάρτηση request_thread() ελέγχει το πεδίο read_or_write της δομής threads_data που δίνεται ως όρισμα.
   ii. Εάν το πεδίο read_or_write είναι 1 (λειτουργία εγγραφής), η συνάρτηση καλεί την _write_test() με τις παραμέτρους count, r και timeflag από τη δομή threads_data. Στη συνέχεια, τερματίζει το νήμα χρησιμοποιώντας την pthread_exit(NULL).
   iii. Εάν η read_or_write είναι 0 (λειτουργία ανάγνωσης), η συνάρτηση καλεί την _read_test() με τις παραμέτρους count, r και timeflag από τη δομή threads_data. Στη συνέχεια, τερματίζει το νήμα χρησιμοποιώντας την pthread_exit(NULL).
   iv. Η συνάρτηση επιστρέφει NULL (αν και αυτή η δήλωση επιστροφής δεν θα εκτελεστεί ποτέ, καθώς η συνάρτηση θα εξέλθει χρησιμοποιώντας την pthread_exit(NULL) και στις δύο περιπτώσεις).

Γενικά: **Η συνάρτηση request_thread() χρησιμεύει ως σημείο εισόδου στο νήμα, εκτελώντας είτε τη συνάρτηση _write_test() είτε τη συνάρτηση _read_test() με βάση το flag read_or_write στη δομή threads_data(t_args). Ουσιαστικά χρησιμεύει ως αποστολέας, κατευθύνοντας το νήμα να εκτελέσει το κατάλληλο benchmark.**

7. Συναρτήσεις _write_test() και _read_test(): Αυτές οι 2 συναρτήσεις ήταν αρχικοποιημένες εξαρχής και εμείς τις τροποποιήσαμε ελάχιστα για τις απαιτήσεις της άσκησης και την σωστή διεκπεραίωση της πολυνηματικής διαδικασίας.

i. Αφαιρέσαμε το άνοιγμα την βάσης μέσα στις συναρτήσεις.

ii. Αφαιρέσαμε τις μεταβλητές για τον υπολογισμό του χρόνου που υπήρχαν καθώς χρησιμοποιούμε δικό μας σύστημα υπολογισμού κόστους όπως αναφέραμε στην συνάρτηση **get_time().** Θα αναλυθεί εκτενέστερα στην πολυνηματική διαδικασία.

```
void _read_test(long int count,int r, int timeflag)
{
    int i;

if (timeflag == 1)
    sum_of_cost[1] = get_time() - sum_of_cost[1];
sum_of_counts[1]+=count;
```

iii. Προσθέσαμε μια ακόμη παράμετρο με όνομα “timeflag” που χρησιμοποιείται για τον τερματισμό της καταγραφής κόστους για το reading ή writing.

iv. Προσθέτει τις διεργασίες στην sum_of_counts αντίστοιχα για το read και write , για να ενημερώνεται ο συνολικός τους αριθμός για την σωστή ενημέρωση στατιστικών.

Γ) THREAD BENHCMARK

8. void cerate thread test(…)»

```
void _create_threads_test(long int count, long int threads, int r, double percentage) {
```

Κάπου εδώ ξεκινάμε την ανάλυση για την πολυνηματική εργασία και τι έχουμε πετύχει σε επίπεδο Threaded Benchmark.

Ο σκοπός της συνάρτησης **_create_threads_test** είναι να δημιουργήσει και να εκτελέσει έναν καθορισμένο αριθμό νημάτων για την εκτέλεση λειτουργιών(benchmark) ανάγνωσης ή/και εγγραφής σε μια βάση δεδομένων. Η συνάρτηση δέχεται παραμέτρους που καθορίζουν τον συνολικό αριθμό των διεργασιών, τον αριθμό των νημάτων, το ποσοστό των διεργασιών εγγραφής και την παράμετρο για τη δημιουργία τυχαίων κλειδιών. Υπολογίζει και **κατανέμει με δίκαιο τρόπο τις διεργασίες** ανάγνωσης και εγγραφής μεταξύ των νημάτων, δημιουργεί τα νήματα, **αναθέτει τις διεργασίες που αναλογούν** κάθε φορά και στη συνέχεια διαχειρίζεται την εκτέλεσή τους. Επιπλέον, η συνάρτηση **μετρά τους**

συνολικούς χρόνους για τα νήματα ανάγνωσης και εγγραφής. Πιο συγκεκριμένα έχουμε:

1. Ανάθεση των παραμέτρων που έχουμε αναφέρει και αναλύσει προηγουμένως.

2. Η συνάρτηση υπολογίζει το συνολικό αριθμό των διεργασιών ανάγνωσης και εγγραφής με βάση την παράμετρο ποσοστό.

3. Στη συνέχεια υπολογίζει τον αριθμό των νημάτων για κάθε λειτουργία (ανάγνωσης και εγγραφής), με βάση την παράμετρο του ποσοστού αν αυτή η υπάρχει και τις αποθηκεύει στον πίνακα που αναφέραμε.

4. Η συνάρτηση δεσμεύει μνήμη για τους πίνακες counts_per_write και counts_per_read για να αποθηκεύσει τον αριθμό των διεργασιών για κάθε νήμα εγγραφής και ανάγνωσης, αντίστοιχα.

5. Μοιράζει ισομερώς τις διεργασίες ανάγνωσης και εγγραφής μεταξύ των αντίστοιχων νημάτων.

Τώρα πάμε να δείξουμε σε βάθος αυτούς τους διαχωρισμούς πριν συνεχίσουμε στα υπόλοιπα βήματα της συνάρτησης.

```
pthread_t* thread_ids = (pthread_t*) malloc(threads * sizeof(pthread_t));
threads_data* t_data = (threads_data*) malloc(threads * sizeof(threads_data));
long int i;
```

a. pthread_t*thread_ids=(pthread_t*)malloc(threads*sizeof(pthread_t)): Θα αποθηκεύει τα αναγνωριστικά των νημάτων που θα δημιουργηθούν. Το μέγεθος του μπλοκ μνήμης που διατίθεται είναι ίσο με το γινόμενο του αριθμού των νημάτων και του μεγέθους της δομής pthread_t.

b. threads_data*t_data=(threads_data*)malloc(threads*sizeof(threads_data)): Αυτή η γραμμή δεσμεύει μνήμη για έναν πίνακα δομών threads_data, οι οποίες θα αποθηκεύουν τα δεδομένα που θα παίρνει κάθε νήμα όπως θα δείξουμε παραάτω. Το μέγεθος του μπλοκ μνήμης που διατίθεται είναι ίσο με το γινόμενο του αριθμού των νημάτων και του μεγέθους τηε δομής threads_data.

c. long int i: Αυτή η γραμμή δηλώνει μια ακέραια μεταβλητή i, η οποία θα χρησιμοποιείται ως μετρητής στους επόμενους βρόχους.

d. Υπολογισμός του αριθμό των διεργασιών ανάγνωσης και εγγραφής με βάση το ποσοστό:

```
// Calculate the number of read and write operations based on the percentage
long int total_write_count = (percentage>=0 && count>2 && percentage!=30000) ? count * percentage / 100 : (count/2);
long int total_read_count = count - total_write_count; //total counts for read
```

i. long int ***total_write_count*** = (percentage>=0 && count>=2 && percentage != 3000) ? count * percentage / 100 : (count/2): Αυτή η γραμμή υπολογίζει τον συνολικό αριθμό των διεργασιών εγγραφής με βάση το δεδομένο ποσοστό. Εάν το ποσοστό είναι μεγαλύτερο ή ισο από 0, έχει τουλάχιστον 2 διεργασίες, υπολογίζει τον αριθμό των εγγραφών πολλαπλασιάζοντας τον συνολικό αριθμό διεργασιών με το ποσοστό και διαιρώντας με το 100. Διαφορετικά, αναθέτει το μισό του συνολικού αριθμού στον αριθμό εγγραφής. (αναφερόμαστε για την readwrite λειτουργία).

ii. long int ***total_read_count*** = count - total_write_count: Αυτή η γραμμή υπολογίζει τον συνολικό αριθμό των λειτουργιών ανάγνωσης αφαιρώντας το σύνολο των διεργασιών από τα δεσμευμένα counts της write λειτουργίας.

e. Κατανομή των νήματα για κάθε λειτουργία:

```
//Calculate the threads for each operation
long int write_threads = (percentage>=0 && threads!=2 && percentage!=30000) ? (threads*percentage/100): (threads/2);
long int read_threads = threads - write_threads ; //total threads for read

threads_for_rw[0] = write_threads; //save the threads for write (for statisticc results)
threads_for_rw[1] = read_threads; //save the threads for read (for statistic results)
```

i. long int ***write_threads*** = (p ercentage>=0 && count>=2 && percentage != 3000) ? threads*percentage/100: (threads/2): Αυτή η γραμμή υπολογίζει τον αριθμό των νημάτων που θα εκτελέσουν λειτουργίες εγγραφής. Εάν το ποσοστό είναι μεγαλύτερο ή ίσο από 0 με τουλάχιστον 2 νήματα, υπολογίζει τα νήματα εγγραφής πολλαπλασιάζοντας τα συνολικά νήματα με το ποσοστό και διαιρώντας με το 100. Διαφορετικά, αναθέτει τα μισά από τα συνολικά νήματα στα νήματα εγγραφής. (readwrite λειτουργία). ***"Εδώ έρχεται η περίπτωση που αν ο χρήστης δώσει 1 thread , τότε θα εκτελέσει μόνο read".***

ii. long int ***read_threads*** = threads - write_threads: Αυτή η γραμμή υπολογίζει τον αριθμό των νημάτων που θα εκτελούν λειτουργίες ανάγνωσης αφαιρώντας τα νήματα εγγραφής από τα συνολικά νήματα.

iii. threads_for_rw[0 ή 1]: Αυτές οι γραμμές αποθηκεύουν για στατιστικούς (***prints_for_results()***) και μόνο σκοπούς τα νήματα που υπολογίστηκαν ομοιόμορφα για κάθε μέθοδο read,write(το είχαμε αναφέραμε και πιο πάνω στην ***bench.h***).

f. Δέσμευση μνήμης για τις διεργασίες ανά νήμα:

i. long int* ***counts_per_write*** = (long int*) malloc(sizeof(long int) * write_threads);: Αυτή η γραμμή δεσμεύει μνήμη για έναν πίνακα ακέραιων αριθμών long που θα αποθηκεύει τον αριθμό των διεργασιών εγγραφής που ανατίθενται σε κάθε νήμα που προορίζεται για την λειτουργίας της εγγραφής.

iv. long int* ***counts_per_read*** = (long int*) malloc(sizeof(long int) * read_threads);: Αυτή η γραμμή δεσμεύει μνήμη για έναν πίνακα ακέραιων αριθμών long που θα αποθηκεύει τον αριθμό των διεργασιών ανάγνωσης που ανατίθενται σε κάθε νήμα ανάγνωσης.

```
//split equal the counts in threads
for(i = 0; i < write_threads; i++){
    counts_per_write[i] = (i < total_write_count % write_threads) ? total_write_count / write_threads + 1 : total_write_count / write_threads;
}
for(i = 0; i < read_threads; i++){
    counts_per_read[i] = (i < total_read_count % read_threads) ? total_read_count / read_threads + 1 : total_read_count / read_threads;
}
```

g. **Μοίρασμα των διεργασιών** ισόποσα μεταξύ των νημάτων:

i. ***Ο πρώτος βρόχος*** for για τα νήματα εγγραφής αναθέτει τον αριθμό των διεργασιών εγγραφής σε κάθε νήμα. Εάν ο τρέχων δείκτης i είναι μικρότερος από το υπόλοιπο της διαίρεσης μεταξύ των συνολικών διεργασιών εγγραφής που υπολογίσαμε προηγουμένως, με τον συνολικό αριθμό των νημάτων εγγραφής που υπολογίσαμε προηγουμένως , τότε αναθέτει μία επιπλέον λειτουργία στο νήμα.

```
counts_per_write[i]=(i<total_write_count%write_threads)?total_write_count/write_threads+1:total_write_count/write_threads;
```

ΣΚΕΨΗ ΠΊΣΩ ΑΠΟ ΑΥΤΗ ΤΗΝ ΠΡΑΞΗ: Σκοπός αυτής της κατανομής είναι η εξισορρόπηση του φόρτου διεργασίας μεταξύ των νημάτων και η αποφυγή σεναρίων όπου ορισμένα νήματα τελειώνουν πολύ νωρίτερα από άλλα, **γεγονός που θα μπορούσε να οδηγήσει σε μη αποδοτική χρήση των πόρων του συστήματος.** Πιο συγκεκριμένα καλύπτονται οι εξής περιπτώσεις:

1. Όταν ο συνολικός αριθμός των διεργασιών εγγραφής <u>διαιρείται ομοιόμορφα</u> με τον αριθμό των νημάτων εγγραφής, κάθε νήμα λαμβάνει ίσο αριθμό διεργασιών.
2. Όταν ο συνολικός αριθμός των διεργασιών εγγραφής <u>δεν διαιρείται ομοιόμορφα</u> με τον αριθμό των νημάτων εγγραφής, οι επιπλέον διεργασιών κατανέμονται ομοιόμορφα στα νήματα με χαμηλότερους δείκτες.

<u>Γενικά</u>: Αυτή η προσέγγιση διασφαλίζει ότι ο <u>φόρτος εργασίας είναι ισορροπημένος μεταξύ των νημάτων</u>, οδηγώντας σε **αποδοτικότερη χρήση των πόρων του συστήματος** και οδηγεί σε **καλύτερες επιδόσεις.**

ii. ***<u>Ο δεύτερος βρόχος</u>*** for επαναλαμβάνει ακριβώς την ίδια σκέψη για τα reads.

Συνεχίζουμε από το νούμερο 5 και πάμε να περιγράψουμε τα υπόλοιπα βήματα για την εκτέλεση των νημάτων για την επίτευξη των διεργασιών.

6. Δημιουργεί τα νήματα και αναθέτει τη διεργασίες ανάγνωσης ή εγγραφής σε κάθε νήμα με βάση την τρέχουσα επιλογή. Στην περίπτωση της επιλογής "readwrite", στα νήματα ανατίθενται εναλλάξ διεργασίες ανάγνωσης και εγγραφής.

7. Η συνάρτηση ορίζει τον χρόνο έναρξης για ανάγνωση και εγγραφή όταν συναντάται το πρώτο νήμα κάθε λειτουργίας. Ορίζει το timeflag για να σταματήσει το ρολόι όταν δημιουργηθούν είτε όλα τα νήματα ανάγνωσης είτε όλα τα νήματα εγγραφής.

8. Για κάθε νήμα, η συνάρτηση ορίζει τις παραμέτρους count, r και read_or_write και δημιουργεί το νήμα χρησιμοποιώντας τη συνάρτηση request_thread. Η συνάρτηση περιμένει να τελειώσουν όλα τα νήματα χρησιμοποιώντας την pthread_join.

9. Τέλος, η συνάρτηση απελευθερώνει την κατανεμημένη μνήμη για τα thread_ids, t_data, counts_per_write και counts_per_read.

Τώρα πάμε να αναλύσουμε σε βάθος αυτές τις λειτουργίες και πως αυτές πραγματεύονται στα σημεία του κώδικα:

READWRITE EXPLANATION

```
// Create threads
for (i = 0; i < threads; i++) {
    if (strcmp(choice, "readwrite") == 0) { //READWRITE
        if ((i % 2 == 0 && write_threads>=1) || (read_threads<=0 && write_threads>=1) ) { //read-write alternate
            if(sum_of_cost[0] == 0) //if this is the first thread write start the clock
                sum_of_cost[0] =get_time();
            t_data[i].count = counts_per_write[write_threads-1]; //pass the counts
            t_data[i].read_or_write = 1; //write flag for request
            write_threads--; //-1 thread for write
        } else { // Assign read operation to the other half of the threads
            if(sum_of_cost[1] == 0) // //if this is the first thread read start the clock
                sum_of_cost[1] =get_time();
            t_data[i].count = counts_per_read[read_threads-1]; //pass the counts
            t_data[i].read_or_write = 0; //set the read flag for request
            read_threads--; //-1 thread for read
        }
```

1) Ο βρόχος ξεκινά με τη ρύθμιση του χρονοδιακόπτη συνολικού κόστους στην τρέχουσα ώρα (get_time()).2
2) Ο κώδικας ελέγχει αν η επιλογή έχει οριστεί σε "readwrite", υποδεικνύοντας ότι οι διεργασίες ανάγνωσης και εγγραφής πρέπει να εκτελούνται παράλληλα.
3) Η εμφωλευμένη συνθήκη if ελέγχει δύο περιπτώσεις:
    a. Εάν ο τρέχων δείκτης i είναι ζυγός και υπάρχουν ακόμη διαθέσιμα νήματα εγγραφής (write_threads >= 1), ο κώδικας αναθέτει μια λειτουργία εγγραφής στο τρέχον νήμα.
    b. Εάν έχουν ανατεθεί όλα τα νήματα ανάγνωσης (read_threads <= 0) και υπάρχουν ακόμη διαθέσιμα νήματα εγγραφής, ο κώδικας αναθέτει μια λειτουργία εγγραφής στο τρέχον νήμα.
4) Εάν στο τρέχον νήμα ανατεθεί μια λειτουργία εγγραφής, ο κώδικας **εκκινεί το χρονόμετρο εγγραφής (εάν είναι το πρώτο νήμα εγγραφής),** αναθέτει τον κατάλληλο αριθμό διεργασιών εγγραφής (counts_per_write[write_threads-1]), θέτει τη σημαία read_or_write σε 1 (υποδεικνύοντας μια λειτουργία εγγραφής) και μειώνει τον αριθμό των διαθέσιμων νημάτων εγγραφής. *Ανατρέχουμε τον πίνακα counts_per_write από το τέλος γιατί δεν μπορούμε να χρησιμοποιήσουμε τον κοινό μετρητή i καθώς κάθε λειτουργία έχει διαφορετικό αριθμό threads.*
5) Εάν στο τρέχον νήμα ανατεθεί μια λειτουργία ανάγνωσης, ο κώδικας ξεκινά το χρονόμετρο ανάγνωσης (εάν πρόκειται για το πρώτο νήμα ανάγνωσης), αναθέτει τον κατάλληλο αριθμό διεργασιών ανάγνωσης (counts_per_read[read_threads-1]), θέτει τη σημαία read_or_write σε 0 (υποδεικνύοντας μια λειτουργία ανάγνωσης) και μειώνει τον αριθμό των διαθέσιμων νημάτων ανάγνωσης. *Ανατρέχουμε τον πίνακα counts_per_read από το τέλος γιατί δεν μπορούμε να χρησιμοποιήσουμε τον κοινό μετρητή i καθώς κάθε λειτουργία έχει διαφορετικό αριθμό threads.*

```
// update the end time
if (write_threads == 0){ //when all write threads are done
    t_data[i].timeflag = 1; //stop the time
    write_threads = -1; //to prevent the if statement
}else if (read_threads == 0){ //the same for read threads
    t_data[i].timeflag = 1;
    read_threads = -1;
}
```

6) Ο κώδικας ενημερώνει τον χρόνο λήξης του χρονοδιακόπτη:
    a. Εάν έχουν εκχωρηθεί όλα τα νήματα εγγραφής, ο κώδικας θέτει τη σημαία **timeflag σε 1** (υποδεικνύοντας τη διακοπή του χρονομετρητή) και θέτει την τιμή write_threads σε -1 για να αποτρέψει να γίνει ξανά αληθής η συνθήκη if.
    b. Εάν έχουν εκχωρηθεί όλα τα νήματα ανάγνωσης, ο κώδικας θέτει το **timeflag σε 1** (υποδεικνύοντας τη διακοπή του χρονομετρητή) και θέτει το read_threads σε -1 για να αποτρέψει τη συνθήκη if από το να είναι ξανά αληθής.

Αρα είδαμε πότε χρησιμοποιείται η σημαία timeflag που βρίσκεται στον structor και πότε ενεργοποείται για την λειτουργία readwrite.

ΣΥΝΟΨΗ: Αυτό το απόσπασμα κώδικα **διασφαλίζει ότι οι λειτουργίες ανάγνωσης και εγγραφής εκτελούνται παράλληλα**, εναλλάσσοντας τα νήματα ανάγνωσης και εγγραφής, **ενώ χειρίζεται επίσης την περίπτωση όπου είτε τα νήματα ανάγνωσης είτε τα νήματα εγγραφής έχουν εξαντληθεί**. Τα χρονόμετρα χρησιμοποιούνται για τη μέτρηση του χρόνου που απαιτείται για τις λειτουργίες ανάγνωσης και εγγραφής ξεχωριστά.

WRITE OR READ EXPLANATION

```
    }
} else { //READ OR WRITE
    t_data[i].count = (i < count % threads) ? count / threads + 1 : count / threads; //split the counts equal to threads for read or w
    t_data[i].read_or_write = (strcmp(choice, "write") == 0) ? 1 : 0; //if write then set flag to 1 for the request else 0 for read
    //time check
    if(t_data[i].read_or_write == 1 && i==0) //if first thread is write then set the time
        sum_of_cost[0] = get_time();
    else if(i==0) //else we have first thread a reader and set the clock
        sum_of_cost[1] = get_time();
    if(i==threads-1) //if the last thread comes then
        t_data[i].timeflag = 1; //stop clock
}
```

Όταν η **μεταβλητή choice** δεν είναι "readwrite", σημαίνει ότι πρέπει να εκτελεστούν είτε λειτουργίες "ανάγνωσης" είτε λειτουργίες "εγγραφής" (όχι και οι δύο). Ας αναλύσουμε τον κώδικα βήμα προς βήμα:

1) Στη μεταβλητή t_data[i].count ανατίθεται ο αριθμός των διεργασιών που πρέπει να εκτελεστούν ανά νήμα. **Ο κώδικας κατανέμει ομοιόμορφα τον συνολικό αριθμό των διεργασιών (count) σε όλα τα νήματα**. Εάν υπάρχει υπόλοιπο, τα πρώτα νήματα λαμβάνουν μία επιπλέον πράξη.
2) Η σημαία t_data[i].read_or_write τίθεται με βάση τη μεταβλητή choice.

a. Εάν η επιλογή είναι "write", η σημαία τίθεται σε 1 (υποδεικνύοντας μια λειτουργία εγγραφής)- διαφορετικά, τίθεται σε 0 (υποδεικνύοντας μια λειτουργία ανάγνωσης).

3)Ο κώδικας ελέγχει αν το τρέχον νήμα είναι το πρώτο νήμα (δηλαδή, i == 0).

a. Εάν πρόκειται για νήμα εγγραφής, ξεκινά ο χρονομετρητής εγγραφής (sum_of_cost[0] = get_time())- διαφορετικά,
b. ξεκινά ο χρονομετρητής ανάγνωσης (sum_of_cost[1] = get_time()).

4) Εάν το τρέχον νήμα είναι το τελευταίο νήμα (i == threads - 1), το t_data[i].timeflag τίθεται σε 1, υποδεικνύοντας τη διακοπή του χρονομετρητή.

```
    t_data[i].r = r; //random keys
    pthread_create(&thread_ids[i], NULL, request_thread, (void*)&t_data[i]); //create thread
}

// join threads
for ( i = 0; i < threads; i++) {
    pthread_join(thread_ids[i], NULL);
}

// free memory
free(thread_ids);
free(t_data);
free(counts_per_write);
free(counts_per_read);
```

Για όλες τις λειτουργίες (read,write,readwrite) αφού η μεταβλητή t_data[i].r πάρει την τιμη του r για τα τυχαία κλειδιά, καλείται η συνάρτηση **pthread_create** για τη **δημιουργία ενός νέου νήματος** με τη συνάρτηση **request_thread** (που αναλύσαμε πιο πάνω) με τα τρέχοντα δεδομένα του νήματος (&t_data[i]). Τέλος κάνουμε join τα νήματα μέχρι να τελειώσουν και αποδεσμεύουμε την μνήνη που δεσμεύσαμε με τις malloc.

ΣΥΝΟΨΗ: Αυτό το απόσπασμα κώδικα δημιουργεί νήματα για την εκτέλεση λειτουργιών ανάγνωσης ή εγγραφής, κατανέμοντας ομοιόμορφα τον συνολικό αριθμό των διεργασιών μεταξύ των νημάτων. Επίσης, **ξεκινάει και σταματάει χρονόμετρα για το πρώτο και το τελευταίο νήμα**, αντίστοιχα, για να μετρήσει το χρόνο που εκτελέστηκαν οι λειτουργίες-μεθοδοι ξεχωριστά.

ΜΕΧΡΙ ΑΥΤΟ ΤΟ ΣΗΜΕΙΟ ΕΧΕΙ ΕΞΑΣΦΑΛΙΣΤΕΊ ΜΕ ΕΠΙΤΥΧΊΑ ΟΛΌΚΛΗΡΟ ΤΟ ΜΈΡΟΣ ΤΗΣ ΑΣΚΗΣΗΣ ΠΟΥ ΑΦΟΡΑ ΤΟ THREADED BENCHMARK ΟΠΩΣ ΖΗΤΕΙΤΑΙ ΑΠΟ ΤΙΣ ΑΠΑΙΤΗΣΕΙΣ:

Threaded Benchmark

- Δημιουργία πολλαπλών νημάτων με χρήση παραμέτρου από τη γραμμή εντολών και δίκαιος διαμοιρασμός των εργασιών στα νήματα
- Εισαγωγή νέας λειτουργίας readwrite όπου εκτεούνται 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

Δ) THREAD-SAFE ENGINE:

Πριν συνεχίσουμε να εξηγούμε την thread-safe engine, να κάνουμε μια ανακεφαλαίωση την thread-benchmark:

Αρχικά ανοίγουμε την βάση δεδομένων στην αρχή του benchmark, θεωρούμε ότι είναι μόνο μία, και την κλείνουμε στο τέλος με σκοπό να εισέλθουν σε αυτή πολλά νήματα για να τεστάρουμε την thread-safe engine και τον αποκλεισμό που έχουμε πετύχει.

Στην συνέχεια χωρίζουμε δίκαια και ομοιόμορφα τα counts στα threads με βάση το ποσοστό , αν αυτό υπάρχει, καλύπτοντας όλες τις περιπτώσεις διαίρεσης των threads/counts ώστε να διασφαλιστεί ότι όλες οι διεργασίες θα εκτελεστούν και πως κανένα από τα νήματα δεν θα έχει περισσότερες διεργασίες από τα υπόλοιπα για κάθε μέθοδο ξεχωριστά. Έτσι όπως αναφέραμε προηγουμένως πετυχαίνουμε **εξισορρόπηση του φόρτου διεργασίας μεταξύ των νημάτων και η αποφυγή σεναρίων όπου ορισμένα νήματα τελειώνουν πολύ νωρίτερα από άλλα, γεγονός που θα μπορούσε να οδηγήσει σε μη αποδοτική χρήση των πόρων του συστήματος.**

Στην συνέχεια υλοποιήσαμε μια μέθοδο για να μετρήσουμε τον χρόνο εκτέλεσης κάθε μεθόδου (read and/or write) ξεχωριστά με ακρίβεια δευτερολέπτων, καθώς αυτά εκτελούνται παράλληλα. Συνεπώς μετράμε από το πρώτο νήμα έως το τέλος του τελευταίου νήματος τόσο για read όσο και για write.

Τέλος αφού διασφαλίσαμε όλα τα παραπάνω, εκτυπώνουμε τα αναλυτικά αποτελέσματα, με τον τρόπο που περιεγράφηκε στην συνάρτηση prints_for_results().

«DB.H»:

Σε αυτόν τον κώδικα ορίσαμε μια απλή δομή κλειδώματος αναγνώστη-γραφέα και στην δομή DB προσθέσαμε το κλείδωμα, ενώ ορίσαμε και κάποιες συναρτήσεις που χρησιμοποιούνται για τον συγχρονισμό της βάσης και θα τις περιγράψουμε αναλυτικότερα στο «db.c». Μια σύντομη επεξήγηση του κώδικα παρέχεται παρακάτω.

1. DB struct: Αυτή η δομή αναπαριστά μια βάση δεδομένων με:

```
typedef struct _db {
    char basedir[MAX_FILENAME+1];
    SST* sst;
    MemTable* memtable;

    pthread_mutex_t readwrite_mutex;
    pthread_cond_t readwrite_cond;
    int num_readers;
    int writer_active;
} DB;
```

a. **basedir**: που αντιπροσωπεύει τον βασικό κατάλογο της βάσης δεδομένων, όπου αποθηκεύονται τα αρχεία δεδομένων.
b. **sst:** Ένας δείκτης σε μια δομή SST (SSTable, ή Sorted String Table), η οποία είναι μια δομή δεδομένων που χρησιμοποιείται στη βάση δεδομένων για την αποθήκευση ταξινομημένων ζευγών κλειδιού-τιμής.

c. **memtable**: Ένας δείκτης σε μια δομή MemTable, η οποία είναι μια δομή δεδομένων στη μνήμη που χρησιμοποιείται στη βάση δεδομένων για την αποθήκευση ζευγών κλειδιών-τιμών πριν από την εγγραφή τους στο δίσκο.
d. **readwrite_mutex**: Χρησιμοποιείται για το συγχρονισμό μεταξύ των νημάτων κατά την πρόσβαση στη βάση δεδομένων.

e. **readwrite_cond:** Μια pthread_cond_t , που χρησιμοποιείται για τη σηματοδότηση και την αναμονή κατά τη διάρκεια των λειτουργιών ανάγνωσης και εγγραφής στη βάση δεδομένων.

f. **num_readers**: Ένας ακέραιος αριθμός που παρακολουθεί τον αριθμό των ενεργών αναγνωστών που έχουν πρόσβαση στη βάση δεδομένων.

g. **writer_active**: Ένας ακέραιος αριθμός που υποδεικνύει αν ένας εγγραφέας είναι ενεργός (1) ή όχι (0).

«DB.C»:

Εδώ θα αναλύσουμε τις συναρτήσεις που ορίσαμε και το πως χρησιμοποιούνται στην πολυνηματική υλοποίηση. Στην συνέχεια θα αναλύσουμε και περιπτώσεις αυτής της πολυνηματικής υλοποίησης περιγράφοντας κυρίως την σχέση μεταξύ readers-writers.

Αρχικά πριν μπούμε στην περιγραφή να περιγράψουμε λίγο 3 συναρτήσεις που είναι χρήσιμες για τα νήματα.

- pthread_cond_broadcast
- pthread_cond_signal
- pthread_cond_wait

1. Οι **pthread_cond_broadcast** και **pthread_cond_signal** είναι δύο συναρτήσεις που χρησιμοποιούνται για να ξεμπλοκάρουν νήματα που περιμένουν σε μια μεταβλητή συνθήκης. Χρησιμοποιούνται σε συνδυασμό με την **pthread_cond_wait**, η οποία επιτρέπει σε ένα νήμα να περιμένει την εκπλήρωση μιας συγκεκριμένης συνθήκης πριν προχωρήσει. Ακολουθεί μια σύντομη επεξήγηση του τρόπου λειτουργίας τους:

   i. pthread_cond_signal: Αυτή η συνάρτηση ξεμπλοκάρει (ξυπνάει) ένα νήμα που περιμένει στη μεταβλητή συνθήκης (θα δούμε παρακάτω για ποια συνθήκη πιο αναλυτικά). Εάν υπάρχουν πολλά νήματα που περιμένουν, η συνάρτηση επιλέγει ένα νήμα για να αφυπνίσει, αλλά η επιλογή δεν καθορίζεται. Σκοπός αυτής της συνάρτησης είναι να ειδοποιήσει ένα μόνο νήμα που περιμένει ότι η συνθήκη που περίμενε έχει εκπληρωθεί ή αλλάξει. Γενικά χρησιμοποιείται όταν θέλουμε να αφυπνίσουμε μόνο ένα νήμα αναμονής, π.χ. όταν θέλουμε να επιβάλλουμε κάποια μορφή προτεραιότητας ή σειράς μεταξύ των νημάτων.

   ii. **pthread_cond_broadcast:** Αυτή η συνάρτηση ξεμπλοκάρει (ξυπνάει) όλα τα νήματα που περιμένουν στη μεταβλητή συνθήκης. Χρησιμοποιείται όταν μια αλλαγή στην κατάσταση του κοινόχρηστου πόρου ή της συνθήκης επηρεάζει όλα τα νήματα που περιμένουν και θέλουμε όλα αυτά τα νήματα να ξανά ελέγξουν τη συνθήκη.

2. Συναρτήσεις:

   Αυτός ο κώδικας υλοποιεί συγχρονισμό με τον κατάλληλο αμοιβαίο αποκλεισμό αναγνώστη-γραφής χρησιμοποιώντας mutexes και μεταβλητές κατάστασης. Ο σκοπός αυτού του κώδικα είναι να επιτρέπει πολλαπλές ταυτόχρονες λειτουργίες ανάγνωσης,

εξασφαλίζοντας ταυτόχρονα αποκλειστική πρόσβαση για λειτουργίες εγγραφής, ώστε να διατηρείται η συνέπεια των δεδομένων.

Πρώτα αρχικοποιούμε τα mutex , τα conditions και τις μεταβλητές του db struct στην συνάρτηση:

DB* db_open_ex(const char* basedir, uint64_t cache_size) - >

```
DB* db_open_ex(const char* basedir, uint64_t cache_size)
{
                                    _

 pthread_mutex_init(&(self->readwrite_mutex), NULL);
 pthread_cond_init(&(self->readwrite_cond), NULL);
 self->num_readers = 0;
 self->writer_active = 0;
```

Και κανουμε destroy στην συνάρτηση: void db_close(DB *self) ->

```
void db_close(DB *self)
-
                                    _
pthread_mutex_destroy(&(self->readwirte_mutex));
pthread_cond_destroy(&(self->readwrite_cond));
```

Α. readers_lock και readers_unlock: Αυτές οι συναρτήσεις είναι υπεύθυνες για την δέσμευση και την αποδέσμευση κλειδαριών ανάγνωσης, αντίστοιχα.

1.1. readers_lock:

```
void readers_lock(DB* self)
{
    pthread_mutex_lock(&self->readwrite_mutex); //lock the mutex
    // wait until no writer is active
    while (self->writer_active) {
        pthread_cond_wait(&self->readwrite_cond, &self->mutex); // wait for the condition to be signal
    }
    self->num_readers++; // increase the number of active readers
    pthread_mutex_unlock(&self->readwrite_mutex); // Unlock the mutex
}
```

- Δεσμεύει readwrite_mutex για να εξασφαλίσει αποκλειστική πρόσβαση.

- Περιμένει να τελειώσει ο εγγραφέας ελέγχοντας την writer_active. Εάν ένας εγγραφέας είναι ενεργός, ο αναγνώστης περιμένει τη μεταβλητή κατάστασης readwrite_cond.

- Αυξάνει τον μετρητή num_readers, υποδεικνύοντας ότι ένας επιπλέον αναγνώστης είναι ενεργός.

- Απελευθερώνει το readwrite_mutex.

1.2. readers_unlock:

```
void readers_unlock(DB* self)
{
    pthread_mutex_lock(&self->readwrite_mutex); // lock the mutex
    self->num_readers--; // decrease the number of active readers
    if (self->num_readers == 0) {
        pthread_cond_signal(&self->readwrite_cond); // signal waiting writers if no readers are active
    }
    pthread_mutex_unlock(&self->readwrite_mutex); // unlock the mutex
}
```

- Δεσμεύει το readwrite_mutex για να εξασφαλίσει αποκλειστική πρόσβαση.

- Μειώνει τον μετρητή num_readers.

- Εάν δεν υπάρχουν άλλοι αναγνώστες (num_readers == 0), σηματοδοτεί τη μεταβλητή κατάστασης readwrite_cond για να ξεμπλοκάρει ενδεχομένως τους αναμένοντες συγγραφείς.

- Απελευθερώνει το readwrite_mutex.

Β. writer_lock και writer_unlock: Αυτές οι συναρτήσεις είναι υπεύθυνες για την δέσμευση και την απελευθέρωση κλειδαριών εγγραφής, αντίστοιχα.

2.1. writer_lock:

```
void writer_lock(DB* self)
{
    pthread_mutex_lock(&self->readwrite_mutex); // lock the mutex
    // Wait until no readers or writers are active
    while (self->num_readers > 0 || self->writer_active) {
        // wait for the condition to be signal
        pthread_cond_wait(&self->readwrite_cond, &self->readwrite_mutex);
    }
    self->writer_active = 1; // activate the writer
    pthread_mutex_unlock(&self->readwrite_mutex); // unlock the mutex
}
```

- Δεσμεύει το readwrite_mutex για να εξασφαλίσει αποκλειστική πρόσβαση.

- Περιμένει να τελειώσουν οι αναγνώστες και οι άλλοι εγγραφείς ελέγχοντας τις num_readers και writer_active. Εάν κάποιοι είναι ενεργοί, ο εγραφέας περιμένει τη μεταβλητή κατάστασης readwrite_cond.

- Θέτει τη σημαία writer_active σε 1, υποδεικνύοντας ότι ένας εγγραφέας είναι ενεργός.

- Απελευθερώνει την readwrite_mutex.

2.2. writer_unlock:

```
void writer_unlock(DB* self)
{
    pthread_mutex_lock(&self->readwrite_mutex); // lock the mutex
    self->writer_active = 0; // deactivate the writer
    pthread_cond_broadcast(&self->readwrite_cond); // broadcast to wake up waiting readers and writ
    pthread_mutex_unlock(&self->readwrite_mutex); // unlock the mutex
}
```

- Δεσμεύει το readwrite_mutex για να εξασφαλίσει αποκλειστική πρόσβαση.

- Επαναφέρει τη σημαία writer_active στο 0.

- Ενημερώνει τη μεταβλητή κατάστασης readwrite_cond για να ξεμπλοκάρει όλους τους αναγνώστες και τους εγγραφείς που περιμένουν.

- Απελευθερώνει το readwrite_mutex.

C. **db_add** και **db_get**: Αυτές οι συναρτήσεις εκτελούν λειτουργίες εγγραφής και ανάγνωσης στη βάση δεδομένων, αντίστοιχα.

3.1. db_add:

```
int db_add(DB* self, Variant* key, Variant* value)
{
    int status;

    writer_lock(self); //call the locker

    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);
    }

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

    writer_unlock(self); //unlock condition
    return status;
}
```

- Καλεί την writer_lock για την ενεργοποίηση κλειδώματος εγγραφής.

- Ελέγχει εάν ο πίνακας μνήμης χρειάζεται συμπίεση και εκτελεί τη συμπίεση εάν είναι απαραίτητο.

- Προσθέτει το ζεύγος κλειδιού-τιμής στον πίνακα μνήμης.

- Καλεί την writer_unlock για να απελευθερώσει το κλείδωμα εγγραφής.

- Επιστρέφει την κατάσταση.

3.2. db_get:

```
int db_get(DB* self, Variant* key, Variant* value)
{
    int status;

    readers_lock(self); //call lock

    if (memtable_get(self->memtable->list, key, value) == 1)
    {
        status = 1;
    }
    else
    {
        status = sst_get(self->sst, key, value);
    }

    readers_unlock(self); //unlock

    return status;
}
```

- Καλεί το readers_lock για να ενεργοποιήσει κλείδωμα ανάγνωσης.

- Προσπαθεί να ανακτήσει την τιμή που σχετίζεται με το κλειδί από τον πίνακα μνήμης. Εάν βρεθεί, η συνάρτηση επιστρέφει την κατάσταση 1.

- Εάν η τιμή δεν βρεθεί στον πίνακα μνήμης, επιχειρεί να την ανακτήσει από τον πίνακα SSTable, ενημερώνοντας ανάλογα την κατάσταση.

- Καλεί την reader_unlock για να απελευθερώσει το κλείδωμα ανάγνωσης.

- Επιστρέφει την κατάσταση της λειτουργίας ανάκτησης.

Δ. Συμπεριφορά και σενάρια πολυνηματικής διαδικασίας:

Παρακάτω παρουσιάζονται τρία σενάρια περιπτώσεων που καταδεικνύουν τη συμπεριφορά της βάσης δεδομένων του key-value store με συγχρονισμό αναγνώστη-εγγραφέα.

Σενάριο 1: Πολλαπλοί ταυτόχρονοι αναγνώστες

Ο αναγνώστης 1 καλεί το db_get για να ανακτήσει μια τιμή για ένα συγκεκριμένο κλειδί.

Ο αναγνώστης 1 δεσμεύει κλείδωμα ανάγνωσης καλώντας το readers_lock. Καθώς δεν υπάρχει ενεργός εγγραφέας, ο αναγνώστης 1 αυξάνει τον μετρητή num_readers και συνεχίζει με τη λειτουργία ανάγνωσης.

Ενώ ο αναγνώστης 1 εξακολουθεί να διαβάζει, ο αναγνώστης 2 καλεί επίσης την db_get.

Ο αναγνώστης 2 δεσμεύει κλείδωμα ανάγνωσης, αυξάνει τον μετρητή num_readers και συνεχίζει με τη δική του λειτουργία ανάγνωσης.

Και οι δύο αναγνώστες εκτελούν τις λειτουργίες ανάγνωσης ταυτόχρονα χωρίς να μπλοκάρει ο ένας τον άλλον.

Μόλις τελειώσουν, ο Αναγνώστης 1 και ο Αναγνώστης 2 απελευθερώνουν τα κλειδώματα ανάγνωσης καλώντας την readers_unlock, μειώνοντας τον μετρητή num_readers και ενδεχομένως σηματοδοτώντας τη μεταβλητή κατάστασης readwrite_cond, εάν δεν υπάρχουν άλλοι ενεργοί αναγνώστες.

Σενάριο 2: Ένας συγγραφέας, χωρίς ενεργούς αναγνώστες

Ο εγγραφέας καλεί την db_add για να προσθέσει ένα ζεύγος κλειδιού-τιμής στη βάση δεδομένων.

Ο εγγραφέας δεσμεύει κλείδωμα εγγραφής καλώντας την writer_lock. Καθώς δεν υπάρχουν ενεργοί αναγνώστες ή εγγραφείς, ο εγγραφέας θέτει τη σημαία writer_active σε 1 και συνεχίζει με την λειτουργία εγγραφής.

Ο εγγραφέας εκτελεί τη λειτουργία εγγραφής, συμπεριλαμβανομένης της προσθήκης του ζεύγους κλειδιού-τιμής στον πίνακα μνήμης και της πιθανής συμπίεσης.

Μόλις ολοκληρωθεί, ο εγγραφέας απελευθερώνει το κλείδωμα εγγραφής καλώντας την writer_unlock, επαναφέροντας τη σημαία writer_active και μεταδίδοντας την ενημέρωση κατάστασης στη μεταβλητή κατάστασης readwrite_cond.

Σενάριο 3: Ο εγγραφέας περιμένει να τελειώσουν οι αναγνώστες

Οι αναγνώστες 1 και 2 δεσμεύουν κλειδαριές ανάγνωσης όπως περιγράφεται στο σενάριο 1 και εκτελούν τις λειτουργίες ανάγνωσης ταυτόχρονα.

Ενώ οι αναγνώστες 1 και 2 είναι ακόμη ενεργοί, ο εγγραφέας καλεί την db_add για να προσθέσει ένα ζεύγος κλειδιού-τιμής στη βάση δεδομένων.

Ο εγγραφέας προσπαθεί να δεσμεύσει κλείδωμα εγγραφής καλώντας την writer_lock. Ωστόσο, λόγω των ενεργών αναγνωστών, ο εγγραφέας μπλοκάρεται και περιμένει στη μεταβλητή συνθήκης readwrite_cond.

Ο αναγνώστης 1 ολοκληρώνει τη λειτουργία ανάγνωσης και απελευθερώνει το κλείδωμα ανάγνωσης. Δεδομένου ότι ο αναγνώστης 2 είναι ακόμα ενεργός, ο εγγραφέας παραμένει μπλοκαρισμένος.

Ο αναγνώστης 2 ολοκληρώνει τη λειτουργία ανάγνωσης και απελευθερώνει το κλείδωμα ανάγνωσης. Καθώς δεν υπάρχουν πλέον ενεργοί αναγνώστες, η μεταβλητή συνθήκης readwrite_cond σηματοδοτείται, επιτρέποντας στον εγγραφέας να αποκτήσει το κλείδωμα εγγραφής και να προχωρήσει με τη λειτουργία εγγραφής.

Ο εγγραφέας εκτελεί τη λειτουργία εγγραφής και απελευθερώνει το κλείδωμα εγγραφής όπως περιγράφεται στο σενάριο 2.

Αυτά τα σενάρια περίπτωσης απεικονίζουν πώς ο συγχρονισμός αναγνώστη-εγγραφέα στη βάση δεδομένων του key-value store **επιτρέπει πολλαπλές ταυτόχρονες λειτουργίες ανάγνωσης**, ενώ **διασφαλίζει την αποκλειστική πρόσβαση για τις λειτουργίες εγγραφής**, διατηρώντας τη συνέπεια των δεδομένων και βελτιώνοντας την απόδοση.

ΠΑΡΑΤΗΡΗΣΗΣ ΓΙΑ ΚΑΛΥΤΕΡΗ ΠΟΛΥΝΗΜΑΤΙΚΗ ΚΑΙ ΕΞΗΓΗΣΗ ΙΔΕΑΣ (ΔΕΝ ΕΧΕΙ ΥΛΟΠΟΙΗΘΕΙ):

Σενάριο για βήμα 3 εκφώνησης: Σε αυτό το τροποποιημένο σενάριο,θα είχαμε δύο ξεχωριστές διαδικασίες για την ανάγνωση και την εγγραφή, με έναν αποκλειστικό εγγραφέα για το SST και έναν άλλο για τον πίνακα μνήμης(memtable). Θα εξηγήσουμε πώς θα λειτουργούσαν αυτές οι διαδικασίες διατηρώντας συγχρονισμό και εξασφαλίζοντας τη συνέπεια των δεδομένων.

Κλειδαριές αναγνώστη-εγγραφέα: Θα εξακολουθήσουμε να χρησιμοποιούμε κλειδαριές αναγνώστη-εγγραφέα για συγχρονισμό, αλλά θα τροποποιήσουμε ελαφρώς τον μηχανισμό κλειδώματος. Θα είχαμε δύο σύνολα κλειδαριών αναγνώστη-εγγραφής, ένα για το SST (SST_readers, SST_writer_active) και ένα άλλο για τον πίνακα μνήμης (Memtable_readers, Memtable_writer_active).

Διαδικασία ανάγνωσης: Όταν ζητείται μια λειτουργία ανάγνωσης, ο αναγνώστης δεσμεύει πρώτα ένα κλείδωμα ανάγνωσης για το SST και στη συνέχεια για τον πίνακα μνήμης(memtable). Ο αναγνώστης θα προσπαθήσει πρώτα να διαβάσει το κλειδί από τον πίνακα μνήμης και αν δεν το βρει, θα προχωρήσει στην ανάγνωση από το SST. Αφού ολοκληρωθεί η λειτουργία ανάγνωσης, ο αναγνώστης θα αποδεσμεύσει και τα δύο κλειδώματα ανάγνωσης με την αντίστροφη σειρά που αποκτήθηκαν (πρώτα για τον πίνακα μνήμης και μετά για το SST).

Διαδικασία εγγραφής: Όταν ζητείται μια λειτουργία εγγραφής, ο εγγραφέας θα καθορίσει πρώτα αν η εγγραφή πρέπει να γίνει στον πίνακα μνήμης ή στο SST. Η απόφαση αυτή μπορεί να βασίζεται σε παράγοντες όπως το μέγεθος του memtable ή το κλειδί που γράφεται.

a. Εάν η λειτουργία εγγραφής αφορά τον πίνακα μνήμης, ο εγγραφέας θα δεσμεύσει κλείδωμα εγγραφής για τον πίνακα μνήμης, θα εκτελέσει τη λειτουργία εγγραφής και στη συνέχεια θα αποδεσμεύσει το κλείδωμα εγγραφής.

b. Εάν η λειτουργία εγγραφής αφορά το SST, ο εγγραφέας θα δεσμεύσει κλείδωμα εγγραφής για το SST και στη συνέχεια για τον πίνακα μνήμης. Αυτό συμβαίνει επειδή οποιαδήποτε λειτουργία συμπίεσης μπορεί να απαιτεί πρόσβαση τόσο στο SST όσο και στον πίνακα memtable. Ο εγγραφέας θα εκτελέσει τη λειτουργία εγγραφής, θα συμπιέσει ενδεχομένως το SST και τον πίνακα memtable και στη συνέχεια θα απελευθερώσει τα κλειδώματα εγγραφής με την αντίστροφη σειρά που αποκτήθηκαν (πρώτα για τον πίνακα memtable και μετά για το SST).

Αυτή η τροποποιημένη διαδικασία, με ξεχωριστούς αποκλειστικούς εγγραφείς για το SST και τον πίνακα μνήμης, εξακολουθεί να εξασφαλίζει τον κατάλληλο συγχρονισμό και τη συνέπεια των δεδομένων. Επιτρέπει ταυτόχρονες λειτουργίες ανάγνωσης και παρέχει αποκλειστική πρόσβαση σε λειτουργίες εγγραφής τόσο στο SST όσο και στον πίνακα μνήμης. Επιπλέον, εξυπηρετεί οποιαδήποτε λειτουργία συμπίεσης που μπορεί να απαιτεί πρόσβαση τόσο στο SST όσο και στον πίνακα μνήμης.

-**Μια άλλη πιθανή υλοποίηση** θα ήταν να κάνουμε αποκλεισμό στην sst_get , sst_merge kai sst_compaction, ώστε όταν θα δεσμεύουν πολλοί την κλειδαριά να σιγουρευόμαστε ότι έχει τελειωσει ολοκληρωτικά η διαδικασία compaction, για να μην να freezarei το σύστημα, δηλαδή έναν αμοιβαίο αποκλεισμό σε αυτές τις συναρτήσεις.

Ε) ΣΤΑΤΙΣΤΙΚΑ ΚΑΙ ΣΧΟΛΙΑΣΜΟΙ ΑΠΟΤΕΛΕΣΜΑΤΩΝ:

Hardware Υλοποίησης πειραμάτων:

Πριν συνεχίσουμε στα στατιστικά των πειραμάτων να σημειωθεί ότι το υλικό μέρος που χρησιμοποιήθηκε είναι το εξής:

- 1 gb ram ddr4,
- 1 cpu core υψηλης συχνότητας,
- 4 gb ssd αποθηκευτικού χώρου.

Σε αυτή την ενότητα θα μελετήσουμε τα στατιστικά αποτελέσματα εκτελώντας τα benchmark για να κατανοήσουμε πόσο βελτιώνει η πολυνηματική διαδικασία τους χρόνους συστήματος, και στην συνέχεια θα σχολιάσουμε τι παρατηρούμε.

1. WRITE STATISTICS:

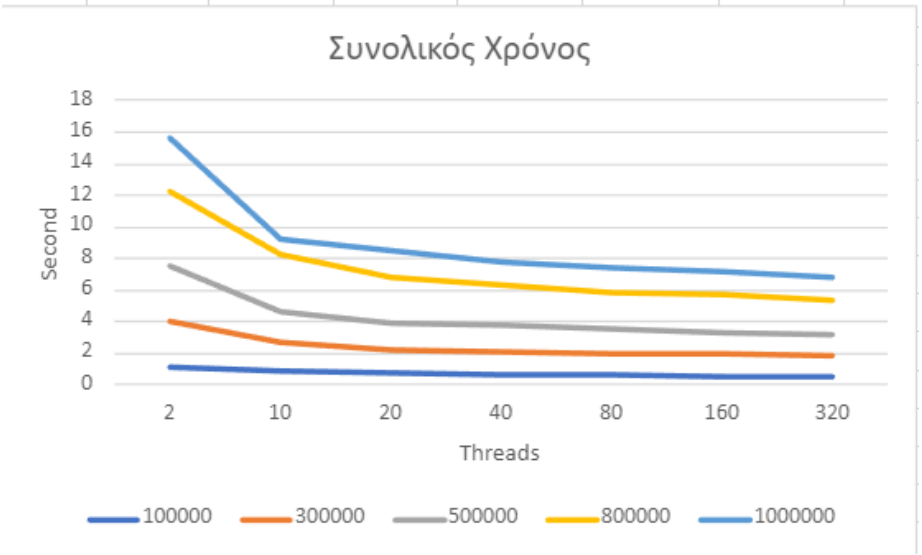
Πρώτα τρέχουμε την κάθε διαδικασία με 1 νήμα ώστε μετά αυξάνοντας τα, να παρατηρήσουμε αν υπάρχει κάποια βελτίωση.

a. 1 thread: counts - > 500000, cost -> 8.10sec

```
------------ RESULTS ------------
a)Writes -> done:500000
b)Seconds/operation -> 0.000016 sec/op
c)Writes/second(estimated) -> 61681.9 writes/sec
d)Threads -> 1
e)Seconds/Thread(estimated) -> 8.106105 sec/th
f)Total cost -> 8.106105 sec
```

b. Many threads diagram for different counts and board with the costs:

Συνολικος χρόνος		Operations				
		100000	300000	500000	800000	1000000
Threads	2	1,131979	4,040047	7,508566	12,303891	15,57616
	10	0,86369	2,6374	4,607598	8,274901	9,260902
	20	0,781548	2,210806	3,938265	6,781828	8,461514
	40	0,694268	2,049487	3,799922	6,327207	7,800414
	80	0,58109	2,006546	3,483693	5,806514	7,405297
	160	0,561721	1,916974	3,304935	5,700853	7,124523
	320	0,551075	1,900178	3,193272	5,37541	6,786715



Παρατήρηση: Παρατηρούμε ότι όσο περισσότερες είναι οι διεργασίες(Οι γραμμές με τα διάφορα χρώματα εκφράζουν πειράματα για διάφορετικό αριθμό διεργασιών) , τόσο ο μέσος χρόνος αυξάνεται. Όμως αυξάνοντας τον αριθμό των threads παρατηρούμε βελτίωση του κόστους στο χρόνο. Το κόστος όπως φαίνεται στο διάγραμμα μετά από ένα αριθμό thread (μετά τα 40) δεν εμφανίζει ιδιαίτερη βελτίωση και παραμένει σχεδόν σταθερός, σε αντίθεση με την αρχική αύξηση των threads. Δηλαδή ο μεγαλυτερος αριθμός thread δεν συνάβει απαραίτητα με την βελτίωση του χρόνου εγγραφής μετά από ένα κατώφλι.

Επίσης γενικά όσο αυξάνονται οι διεργασίες αυξάνεται και ο χρόνος εκτέλεσης του benchmark. Όπως βλέπουμε στον πίνακα , όσο περισσότερα τα counts τόσο μεγαλύτεροι και οι χρόνοι για το wirte. Απλώς η ιδέα και η παρατήρηση βασίζεται στο πώς τα threads βελτιώνουν το κόστος σε χρόνο.

Σχολιασμός: Αρχικά πριν σχολιάσουμε να εξηγήσουμε τι συμβαίνει συνοπτικά στην εγγραφή και ποιοι παράγοντες επηρεάζουν τον χρόνο.

α. **Threads and the lock we implemented**: Όταν πολλά νήματα εγγραφής έχουν ταυτόχρονη πρόσβαση στο δέντρο LSM, πρέπει να αποκτήσουν το ενιαίο κλείδωμα για να εκτελέσουν λειτουργίες εγγραφής. Αυτό το κλείδωμα επιβάλλει αμοιβαίο αποκλεισμό, διασφαλίζοντας ότι μόνο ένα νήμα μπορεί να γράφει κάθε φορά, οδηγώντας σε ανταγωνισμό μεταξύ των νημάτων. Καθώς αυξάνεται ο αριθμός των νημάτων, αυξάνεται και ο ανταγωνισμός για το κλείδωμα, με αποτέλεσμα τα νήματα να ξοδεύουν περισσότερο χρόνο περιμένοντας να απελευθερωθεί το κλείδωμα. Όμως χωρίζοντας ομοιόμορφα τις διεργασίες στα νήματα γίνεται και πιο γρήγορα η κάθε εγγραφή στο memtable και sst με τον τρόπο που θα περιγραφεί αμέσως μετά αλλά και πως ο μεγάλος αριθμός thread το επηρεάζει.

β. **Memtable και SST:** Η δομή δεδομένων LSM-tree αποτελείται από έναν memtable και έναν πολυεπίπεδο πίνακα ταξινομημένων συμβολοσειρών (SST). Οι λειτουργίες εγγραφής εγγράφονται πρώτα στον memtable στη μνήμη. Μόλις ο πίνακας μνήμης φθάσει σε ένα ορισμένο μέγεθος, εκκαθαρίζεται στον SST και μια διαδικασία συμπίεσης συγχωνεύει τα δεδομένα από διαφορετικά επίπεδα. Με πολλαπλά νήματα που εκτελούν λειτουργίες εγγραφής, ο πίνακας μνήμης μπορεί να γεμίσει πιο γρήγορα, προκαλώντας πιο συχνές εκκαθαρίσεις και συμπυκνώσεις. Αυτό μπορεί να επηρεάσει τη συνολική απόδοση εγγραφής, καθώς η συμπύκνωση είναι μια διαδικασία έντασης εισόδου/εξόδου.

γ. **Hardware (Υλικό)**: Οι πόροι υλικού, όπως η CPU, η μνήμη και ο αποθηκευτικός χώρος, παίζουν επίσης κρίσιμο ρόλο στον καθορισμό της απόδοσης του LSM-tree με πολλά νήματα εγγραφής. Καθώς αυξάνεται ο αριθμός των νημάτων, αυξάνεται και η ζήτηση για πόρους CPU, μνήμης και εισόδου/εξόδου. Όταν οι πόροι υλικού κορεστούν, η προσθήκη περισσότερων νημάτων δεν οδηγεί σε καλύτερες επιδόσεις. Ίσα ίσα προκαλέι segfault το οποίο λύνεται με την αύξηση των πόρων, κυρίως στην μνήμη ram. Επιπλέον, η απόδοση του υποσυστήματος αποθήκευσης, ιδίως κατά τη συμπίεση, μπορεί να αποτελέσει κρίσιμο σημείο αναφοράς στη συνολική απόδοση εγγραφής.

Εξήγηση:

Λαμβάνοντας υπόψη αυτούς τους παράγοντες, η παρατηρούμενη συμπεριφορά μπορεί να εξηγηθεί ως εξής: Αρχικά, η αύξηση του αριθμού των νημάτων εγγραφής βελτιώνει την απόδοση λόγω της καλύτερης αξιοποίησης των πόρων υλικού και του αυξημένου παραλληλισμού. Ωστόσο, καθώς ο αριθμός των νημάτων συνεχίζει να αυξάνεται, ο ανταγωνισμός για το μοναδικό κλείδωμα γίνεται περιοριστικός παράγοντας, με αποτέλεσμα τα νήματα να ξοδεύουν περισσότερο χρόνο περιμένοντας το κλείδωμα. Ταυτόχρονα, η αυξημένη συχνότητα των εκκαθαρίσεων των memtable και των συμπυκνώσεων SST λόγω των πολλαπλών νημάτων επίσης συμβάλει στην επιβράδυνση της απόδοσης εγγραφής. Τέλος, ο κορεσμός των πόρων υλικού περιορίζει τις δυνατότητες περαιτέρω βελτίωσης της απόδοσης, οδηγώντας σε φθίνουσες αποδόσεις πέραν ενός ορισμένου αριθμού νημάτων.

Συμπέρασμα: Σε αυτό το σενάριο, η εύρεση μιας βέλτιστης ισορροπίας μεταξύ του αριθμού των νημάτων, της δομής δεδομένων LSM-tree και των διαθέσιμων πόρων υλικού αποτελεί σημαντικό κομμάτι για την ομαλή επίτευξη της καλύτερης απόδοσης εγγραφής.

Βελτιωμένο σενάριο εγγραφής (θεωρητικά): Τώρα για το θεωρητικό σενάριο εγγραφής που αναφέραμε στην πολυνηματική διαδικασία όπου έχουμε 2 εγγραφείς παράλληλα (έναν στο memtable και έναν στο sst), μπορεί να βελτιώσει επιπλεόν τον χρόνο εγγραφής , αλλά η διαφορά θα είναι ελάχιστη. Αυτό οφείλεται στο γεγονός ότι η εγγραφή στο memtable είναι πολύ γρήγορη και ίδια σε κόστος για κάθε thread εγγραφής λόγω της ομοιόμορφης διασποράς των διεργασιών. Αυτό όμως σημαίνει ότι στην κλειδαρία του sst θα περιμένανε πολλά νήματα διότι η διαδικασία εγγραφής στο sst είναι κοστοβόρα λόγω των πολλών επιπέδων και της συγχώνευσης. Άρα υπάρχει ένα trade-off σενάριο για αυτής της κελιδαρίάς. **Για να πετύχουμε τους καλύτερους δυνατούς χρόνους** θα έπρεπε τα αφήσουμε

πολλά νήματα παράλληλα στο sst βάζοντας κλειδαριες στα διάφορα επιπέδα compaction δημιουργόντας τον κατάλληλο αποκλεισμό.

2. READ STATISTICS:

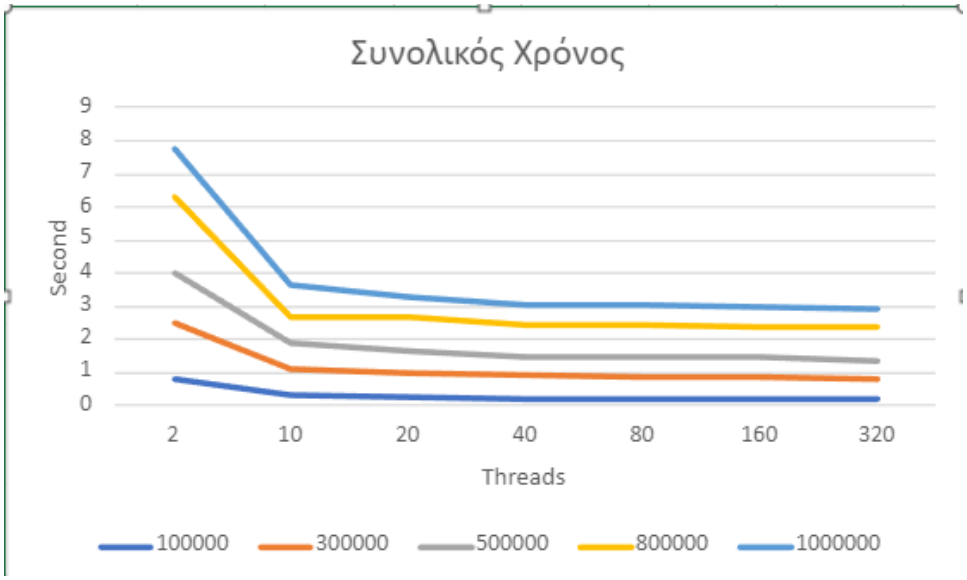
Τα στατιστικά για το read πραγματοποιήθηκαν μετά από τις αντίστοιχες εγγραφές.

a . 1 thread: counts -> 500000 , costs - > 5.97sec

```
------------ RESULTS ------------
a)Reads -> done:500000
b)Found -> 500000
c)Seconds/operation -> 0.000012 sec/op
d)Reads/second(estimated) -> 83732.2 read/sec
e)Threads -> 1
f)Seconds/Thread(estimated) -> 5.971417 sec/th
g)Total cost -> 5.971417 sec
```

b. **Many threads diagram for different counts and board with the costs:**

Συνολικος χρόνος		Operations				
		100000	300000	500000	800000	1000000
Threads	2	0,783147	2,485471	4,010676	6,295166	7,759787
	10	0,331852	1,113359	1,896992	2,688444	3,62672
	20	0,24235	0,994739	1,680447	2,682223	3,2694
	40	0,228724	0,897267	1,482404	2,406789	3,064842
	80	0,220783	0,882989	1,466355	2,406528	3,041552
	160	0,210026	0,834972	1,458751	2,400387	2,961291
	320	0,200652	0,800361	1,354714	2,372426	2,891767



Παρατήρηση: Εδώ οι χρόνοι του read παρουσιάζονται με τον ίδιο σχεσιακό μοντελο πίνακα τιμών - διάγραμμα. Εδώ όπως εύκολα παρατηρούμε οι χρόνοι του read είναι με διαφορά πιο γρήγοροι από τους χρόνους εγγραφής στα ίδια count και threads. Παρόλα αυτά και εδώ παρατηρούμε ένα άνω όριο συνολικών threads για την βελτίωση του χρόνου. Μετά από ένα σημείο όσα threads και αν βάλουμε ο χρόνος παραμένει σχεδόν σταθερός και αμετάβλητος σε σχέση με την αρχική αύξηση των threads που μας δίνει ραγδαία μείωση του χρόνου. Ακολουθεί εξήγηση των παρατηρήσεων.

Και εδώ όσο αυξάνονται οι διεργασίες αυξάνεται και ο χρόνος εκτέλεσης του benchmark. Όπως βλέπουμε στον πίνακα , όσο περισσότερα τα counts τόσο μεγαλύτεροι και οι χρόνοι για το read . Απλώς η ιδέα και η παρατήρηση βασίζεται στο πώς τα threads βελτιώνουν το κόστος σε χρόνο.

Εξήγηση:

- Οι λειτουργίες ανάγνωσης είναι γενικά ταχύτερες από τις λειτουργίες εγγραφής: Ενώ οι λειτουργίες εγγραφής απαιτούν ενημέρωση ή τροποποίηση των δομών δεδομένων, όπως συμπιέσεις και εκροές στο SST.
- Υλοποίηση κλειδώματος πολλαπλών αναγνωστών και ενός εγγραφέα: Η παρεχόμενη υλοποίηση κλειδώματος επιτρέπει πολλαπλές ταυτόχρονες λειτουργίες ανάγνωση. Αυτό σημαίνει ότι η συνάρτηση readers_lock επιτρέπει σε πολλαπλούς αναγνώστες να έχουν ταυτόχρονη πρόσβαση στα δεδομένα. Αυτή η υλοποίηση παρέχει ένα επίπεδο παραλληλισμού στις λειτουργίες ανάγνωσης, επιτρέποντας ταχύτερους χρόνους ανάγνωσης.
- Περιορισμοί υλικού: Καθώς αυξάνεται ο αριθμός των νημάτων, το σύστημα μπορεί να αντιμετωπίσει περιορισμούς που επιβάλλονται από το υλικό ή το λογισμικό, όπως ο αριθμός των διαθέσιμων πυρήνων της CPU , την ραμ και τον μέγιστο αριθμό ταυτόχρονων νημάτων που υποστηρίζονται από το λειτουργικό μας σύστημα. Πέρα από ένα ορισμένο σημείο, η προσθήκη περισσότερων νημάτων ενδέχεται να μην οδηγήσει σε περαιτέρω βελτίωση της απόδοσης λόγω αυτών των περιορισμών.
- Αυξημένος ανταγωνισμός : Καθώς αυξάνεται ο αριθμός των νημάτων, αυξανεται ο ανταγωνισμός για κοινόχρηστους πόρους, όπως τις κλειδαριές, την μνήμη και είσοδος/έξοδος.
- Νήματα - Διεργασίες: Τα νήματα όπως αναφέραμε έχουν ομοιόμορφο αριθμό διεργασιών. Όταν αυξάνεται ο αριθμός των νημάτων, το

όφελος από την προσθήκη περισσότερων νημάτων μειώνεται, καθώς κάθε πρόσθετο νήμα έχει την ίδια "εργασία" να κάνει ή μπορεί να χρειαστεί να περιμένει για κοινόχρηστους πόρους. Αυτό έχει ως αποτέλεσμα μια φθίνουσα απόδοση στη βελτίωση των επιδόσεων, οδηγώντας στο κάτω όριο που παρατηρείται στους χρόνους ανάγνωσης.

- Τέλος να σημειωθεί ότι ο χρόνος ανάγνωσης επηρεάζεται αν ένα νήμα χρειάστει να ψάξει σε κάποιο επίπεδο του sst. Τότε ο χρόνος αυξάνεται καθώς η ανάγνωση στον πίνακα μνήμης είναι πιο γρήγορη από την ανάγνωση στα 6 επίπεδα του sst.

Συμπέρασμα: Συμπερασματικά, η παρατηρούμενη συμπεριφορά στους χρόνους ανάγνωσης μπορεί να αποδοθεί στο εγγενές πλεονέκτημα ταχύτητας των λειτουργιών ανάγνωσης, στον παραλληλισμό που επιτρέπει η υλοποίηση της κλειδαριάς για τις λειτουργίες ανάγνωσης και στους περιορισμούς που επιβάλλονται από το υλικό και τον ανταγωνισμό για κοινόχρηστους πόρους καθώς αυξάνεται ο αριθμός των νημάτων. Αυτοί οι παράγοντες συμβάλλουν στο κατώφλι που παρατηρείται στους χρόνους ανάγνωσης μετά από έναν ορισμένο αριθμό νημάτων.

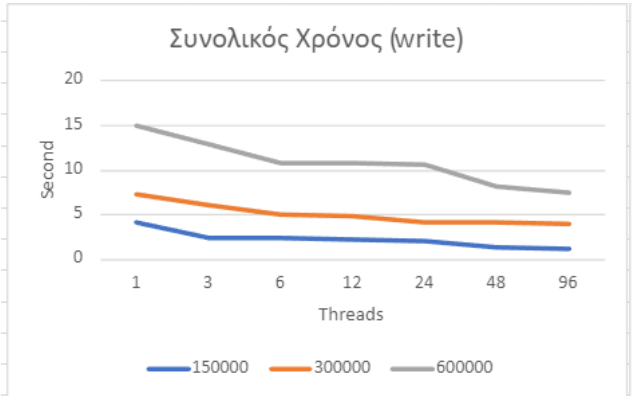
3. READWRITE STATISTICS:

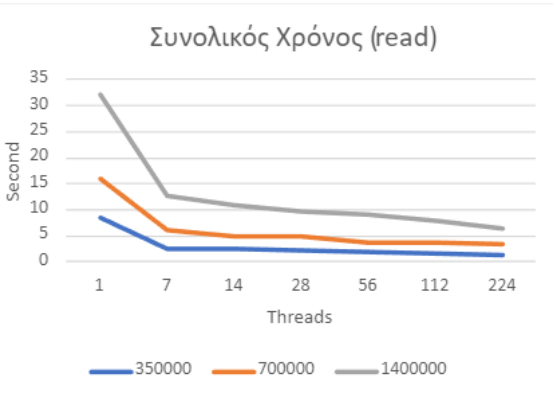
Εισαγωγή: Εδώ δουλέψαμε με διάφορο αριθμό διεργασιών αυξάνοντας τα νήματα , όπως στις προηγούμενες υλοποιήσης με την διαφορά ότι έχουμε εκτελε΄σει benchmarks για ποσοστά(30%,50%,70%) για το σύνολο διεργασιων του write και αντίστοιχα με βάση το ποσοστό αυτό , τις διεργασίες για το read. Θα παρουσιάσουμε τους χρόνους readwrite ξεχωριστά για το read και για το write και πόσα counts, threads χρησιμοποίησε η κάθε μεθοδος ξεχωριστά από τα συνολικά.

a. Percentage 30%

30% readwrite	*Write*		
Συνολικος χρόνος	Operations		
Threads	150000	300000	600000
1	4,2203	7,4043	14,9461
3	2,4681	6,0755	12,9264
6	2,4444	5,0341	10,8722
12	2,3205	4,9138	10,7808
24	2,0609	4,1668	10,5884
48	1,4524	4,2266	8,2411
96	1,1291	3,9971	7,4126

30% readwrite	*Read*		
Συνολικος χρόνος	Operations		
Threads	350000	700000	1400000
1	8,4915	15,9663	32,2011
7	2,4676	6,1224	12,8057
14	2,3394	4,8961	10,8421
28	2,1639	4,8786	9,5329
56	1,8289	3,7148	9,1674
112	1,6004	3,5433	7,8087
224	1,3219	3,4211	6,4994





<u>Πως είναι χωρισμένα τα δεδομένα συνοπτικά στους πίνακες:</u>

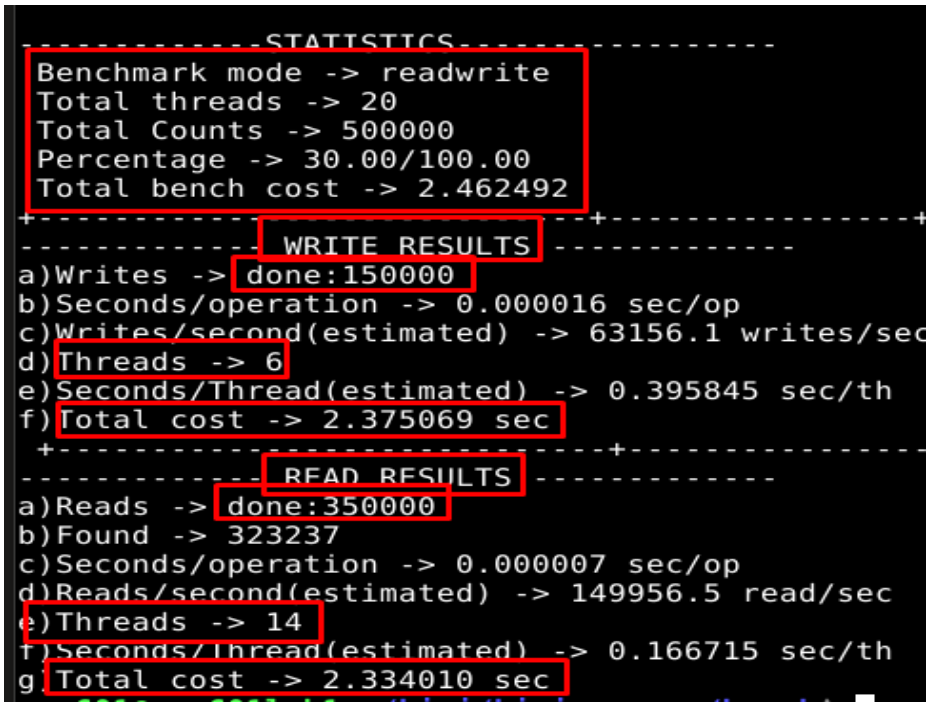
- ***./kiwi-bench readwrite 500000 20 30:***

Πχ: Ποσοστό -> 30% , counts -> 500000, threads -> 20:

- Πίνακας write:Βάση ποσοστού 6 threads ,150000 counts, 2,44sec cost
- Πίνακας read:Βάση ποσοστού 14 threads, 350000 counts, 2,46sec cost
- Σύνολο readwrite: 20 threads 500000 counts, 2,46sec cost

Όπως παρατηρούμε ο συνολικός χρόνος σε κόστος του readwrite είναι περίπου ίσος με τον μεγαλύτερο χρόνο του read ή του write καθώς τρέχουν παράλληλα και συγχρονισμένα και όχι αθροιστικά.

<u>Παράδειγμα Εκτύπωσης στην οθόνη</u>:



<u>Παρατήρηση:</u> Εδώ παρατηρούμε ότι οι χρόνοι κάθε διαδικασίας ξεχωριστά βελτιώνονται καθώς αυξάνουμε τα νήματα, άρα βελτιώνεται και ο συνολικός

χρόνος του benchmark (readwrite). Να σημειωθεί ότι ο χρόνος όπως περιγράψαμε στις προηγούμενες ενότητες καταγράφεται με παράλληλο τρόπο για να έχουμε απόλυτη ακρίβεια. Πάλι όμως έχουμε ένα άνω όριο βελτίωσης κόστους για κάθε διαδικασία. Στην συνέχεια θα εξηγήσουμε λίγο πως λειτουργεί η σχέση μεταξύ τους και πώς αυτό επηρεάζει το αποτέλεσμα στο κόστος και που πιθανολογείται ο αποδοτικότερος χρόνος των read διεργασιών παρόλο που υπερτερούν αριθμητικά.

Αντίστοιχα όσο αυξάνονται οι διεργασίες αυξάνεται και ο χρόνος εκτέλεσης του benchmark. Όπως βλέπουμε στον πίνακα , όσο περισσότερα τα counts τόσο μεγαλύτεροι και οι χρόνοι , τοσο για το read , όσο και για το wirte. Απλώς η ιδέα και η παρατήρηση βασίζεται στο πώς τα threads βελτιώνουν το κόστος σε χρόνο. **Αυτό ισχυεί για όλα τα benchmarks**

Εξήγηση:

Στο συγκεκριμένο σενάριο, οι λειτουργίες εγγραφής χρειάζονται σχεδόν τον ίδιο χρόνο με τις λειτουργίες ανάγνωσης όσο αυξάνουμε τα threads, παρόλο που η ανάγνωση έχει μεγαλύτερο ποσοστό των συνολικών διεργασιών και νημάτων (70%) σε σύγκριση με την εγγραφή (30%). Αυτή η συμπεριφορά μπορεί να αποδοθεί στους ακόλουθους παράγοντες:

- Δέσμευση κλειδώματος λειτουργίας εγγραφής: Η λειτουργία εγγραφής απαιτεί αποκλειστικό κλείδωμα στη δομή δεδομένων και μόνο ένας εγγραφέας μπορεί να εκτελείται ταυτόχρονα. Αυτό σημαίνει ότι όσο αυξάνεται ο αριθμός των νημάτων εγγραφής, ο ανταγωνισμός για το κλείδωμα της λειτουργίας εγγραφής γίνεται πιο σημαντικός, με αποτέλεσμα ορισμένα νήματα εγγραφής να περιμένουν τη σειρά τους. Αντίστοιχα αυξάνεται και ο χρόνος αναμονής και από τα νήματα των αναγνωστών καθώς όσο αυτά εκτελούνται και οι λειτουργίες εγγραφής περιμένουν να ολοκληρωθούν τα πιθανά reads μέχρι να αποδεσμεύσουν τις κλειδαριές.
- Παραλληλισμός λειτουργίας ανάγνωσης: Από την άλλη πλευρά, οι λειτουργίες ανάγνωσης επιτρέπεται να εκτελούνται ταυτόχρονα, καθώς πολλοί αναγνώστες μπορούν να έχουν ταυτόχρονη πρόσβαση στη δομή δεδομένων. Αυτός ο παραλληλισμός στις λειτουργίες ανάγνωσης συμβάλλει στη διατήρηση της απόδοσης ανάγνωσης παρά τον υψηλότερο αριθμό νημάτων ανάγνωσης. Επίσης ο χρόνος ανάγνωσης πραγματεύεται σε γρηγορότερο ρυθμό καθώς υπάρχει πιθανότητα να βρήκε κατευθείαν τα key-value αποθηκευμένα στο memtable και να μην έψαξε στο sst για όλες τις τιμές.

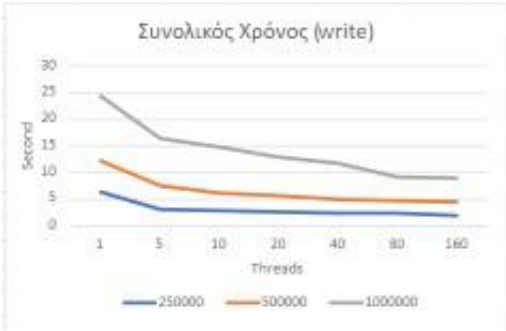
- Ενίσχυση εγγραφής LSM-tree: Όπως αναφέραμε και στο προηγούμενο bullet έμμεσα, οι δομές LSM-tree, όπως αυτή που χρησιμοποιείται , περιλαμβάνουν πολλαπλά επίπεδα οργάνωσης δεδομένων (memtable και SST). Οι λειτουργίες εγγραφής ενδέχεται να απαιτούν πρόσθετη εργασία, όπως συμπίεση, καθώς τα δεδομένα μεταφέρονται από τον πίνακα μνήμης στον SST. Αυτό μπορεί να συμβάλει στον σχετικά μεγαλύτερο χρόνο που απαιτείται για τις λειτουργίες εγγραφής.

Συμπέρασμα: Συμπερασματικά, η παρατηρούμενη συμπεριφορά των λειτουργιών εγγραφής που διαρκούν σχεδόν τον ίδιο χρόνο με τις λειτουργίες ανάγνωσης μπορεί να αποδοθεί σε παράγοντες όπως ο ανταγωνισμός κλειδώματος, ο παραλληλισμός που επιτρέπεται για τις λειτουργίες ανάγνωσης, τις δομές στην οποία έκανε αναζήτηση και την πιθανή σύμπτυξη της δομής του δέντρου LSM. Σε γενικά επίπεδα ο συνολικός χρόνος ανάγνωσης-εγγραφής βελτιώνεται μέχρι ένα ανώτατο όριο thread που υποστηρίζει το σύστημα.

b. Percentage 50%:

50% readwrite	Write		
Συνολικος χρόνος	Operations		
Threads	250000	500000	1000000
1	6,3138	12,1224	24,4151
5	3,1506	7,5241	16,3323
10	2,7794	6,1818	14,8286
20	2,6795	5,7355	12,9948
40	2,4687	4,9273	11,7797
80	2,2722	4,7093	9,2201
160	1,9679	4,5421	9,0202

50% readwrite	Read		
Συνολικος χρόνος	Operations		
Threads	250000	500000	1000000
1	6,3141	12,1226	24,4154
5	3,0281	7,545	15,8094
10	2,6568	5,9166	13,9397
20	2,3878	5,2566	11,8431
40	1,6631	4,7641	9,9522
80	1,2667	4,1621	7,3297
160	0,7739	3,4129	6,3411



Τα διαγράμματα και οι πίνακες παρουσιάζονται με παρόμοιο τρόπο όπως στο προηγούμενο ποσοστό.

Παρατήρηση: Εδώ παρατηρούμε ότι για ομοιόμορφη κατανομή διεργασιών νημάτων η read πάλι εμφανίζει ελάχιστα αποδοτικότερους χρόνους από την write. Σε γενικά πλαίσια παρατηρείται η βελτίωση του χρόνου μέχρι ένα κατωφλι όπως και στις προηγούμενες παρατηρήσεις benchmark που έχουμε κάνει. Ακολουθεί μια πιο συνοπτική από δω και πέρα εξήγηση καθώς έχουμε

περιγράψει αναλυτικά όλους τους παράγοντες που συνδράμουν στο benchmark.

Εξήγηση:

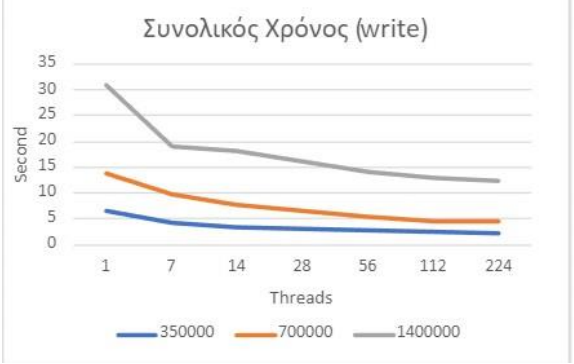
- Εδώ οι χρόνοι αναμονής και δέσμευσης κλειδαριών μεταξύ των Threads με βάση τον υλοποιημένο αποκλεισμό που περιγράψαμε πιθανότατα να είναι ομοιόμορφα κατανεμημένοι καθώς οι διεργασίες έχουν παρόμοιο αριθμό threads. Δηλαδή σε αυτή την περίπτωση πιθανολογείται ότι τα write thread εκτελέστηκαν με μεγαλύτερη προτεραιότητα από τα read thread, με αποτέλεσμα οι ανγνώστες να περιμέναν περισσότερο χρόνο την αποδέσμευση κλειδαριάς.
- Επίσης ένας άλλος καταλυτικός παράγοντας που δεν έκανε τα read να ξεχωρίσουν σε απόδοση από τα write ,όπως στις προηγούμενες περιπτώσεις, είναι να χρειάστηκαν να αναζητήσουν κάποιο κλειδί και στο sst επίπεδο με αποτέλεσμα να αυξηθεί ελαφρώς το κόστος.

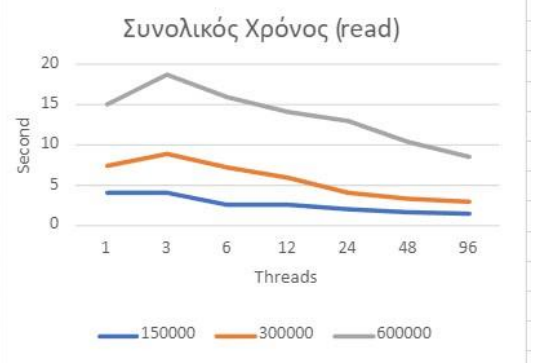
Συμπέρασμα: η παρατηρούμενη συμπεριφορά των λειτουργιών εγγραφής που διαρκούν σχεδόν τον ίδιο χρόνο με τις λειτουργίες ανάγνωσης μπορεί να αποδοθεί σε παράγοντες όπως ο ανταγωνισμός κλειδώματος, ο παραλληλισμός που επιτρέπεται για τις λειτουργίες ανάγνωσης, τις δομές στην οποία έκανε αναζήτηση και την πιθανή σύμπτυξη της δομής του δέντρου LSM. Σε γενικά επίπεδα ο συνολικός χρόνος ανάγνωσης-εγγραφής βελτιώνεται μέχρι ένα ανώτατο όριο thread που υποστηρίζει το σύστημα.

c. Percentage 70%:

70% readwrite	*Write*		
Συνολικος χρόνος	Operations		
Threads	350000	700000	1400000
1	6,6489	13,7372	30,8007
7	4,2861	9,7784	19,0305
14	3,2959	7,7784	18,1958
28	3,1771	6,6178	15,9955
56	2,6551	5,4397	14,0677
112	2,4291	4,4867	13,0129
224	2,2264	4,5562	12,2327

70% readwrite	*Read*		
Συνολικος χρόνος	Operations		
Threads	150000	300000	600000
1	4,1382	7,4923	15,0839
3	4,0465	8,9991	18,7275
6	2,6366	7,2212	15,9962
12	2,5524	5,8782	14,1913
24	2,0249	4,0422	13,0311
48	1,5841	3,3057	10,4318
96	1,4543	2,9822	8,5749





Παρατήρηση: Είναι προφανές ότι οι λειτουργίες εγγραφής απαιτούν περισσότερο χρόνο σε σύγκριση με τις λειτουργίες ανάγνωσης, καθώς ένα μεγαλύτερο ποσοστό των διεργασιών (70%) κατανέμεται σε λειτουργίες εγγραφής. Ωστόσο, εξακολουθεί να υπάρχει αξιοσημείωτη βελτίωση του χρόνου απόδοσης. Είναι ενδιαφέρον ότι με 10 νήματα στο benchmark, υπάρχει μια μικρή αύξηση του χρόνου ανάγνωσης, η οποία μπορεί να αποδοθεί στους παράγοντες που συζητήθηκαν στις προηγούμενες παρατηρήσεις. Παρακάτω παρέχεται μια απλοποιημένη εξήγηση, καθώς η υποκείμενη έννοια παραμένει η ίδια.

Εξήγηση:

- Όπως συζητήθηκε προηγουμένως και λαμβάνοντας υπόψη τη δομή του δέντρου LSM, οι λειτουργίες εγγραφής είναι γενικά πιο αργές από τις λειτουργίες ανάγνωσης. Ο χρόνος επεξεργασίας για τα νήματα εγγραφής επηρεάζεται σημαντικά από το υψηλό ποσοστό διεργασιών που προορίζονται για λειτουργίες εγγραφής. Αυτό έχει ως αποτέλεσμα μια σημαντική διαφορά στο χρόνο σε σύγκριση με τις λειτουργίες ανάγνωσης, οι οποίες έχουν λιγότερες διεργασίες που διατίθενται και είναι εγγενώς ταχύτερες.
- Στο συγκεκριμένο σενάριο, με κατανομή του 70% των διεργασιών σε λειτουργίες εγγραφής, τα νήματα εγγραφής αντιμετωπίζουν μεγαλύτερο ανταγωνισμό για το αποκλειστικό κλείδωμα, γεγονός που οδηγεί σε μεγαλύτερο χρόνο αναμονής και πιο αργές λειτουργίες εγγραφής. Αντίθετα, οι λειτουργίες ανάγνωσης, με κατανομή μόνο του 30% των διεργασιών, αντιμετωπίζουν μικρότερο ανταγωνισμό και μπορούν να εκτελεστούν ταχύτερα λόγω της ταυτόχρονης φύσης του κοινού κλειδώματος. Αυτό έχει ως αποτέλεσμα τις παρατηρούμενες διαφορές στο χρόνο μεταξύ των λειτουργιών εγγραφής και ανάγνωσης.

ΑΝΑΓΡΑΦΗ ΕΚΒΑΣΗΣ ΓΙΑ ΤΟ ΠΙΘΑΝΟ ΣΕΝΑΡΙΟ ΌΠΩΣ ΠΕΡΙΓΡΑΦΘΗΚΕ ΣΤΗΝ ΠΟΛΥΝΗΜΑΤΙΚΗ ΜΕ 2 ΚΛΕΙΔΑΡΙΕΣ ΓΙΑ WRITE:

Λοιπόν σε μια τέτοια υλοποίηση δεν θα έπαιζε σημαντικό ρόλο στην βελτίωση χρόνου των writes για τον ίδιο ακριβώς λόγο που αναφέραμε στην περιγραφική διαδικασία στατιστικών του write (νούμερο 1). Συνεπώς πιθανότατα να υπήρχε και trade-off στο χρόνο που κάνουν και οι λειτουργίες read καθώς θα είχαν και αυτά περισσότερο χρόνο αναμονής αν όχι τον ίδιο με βάση τον αποκλεισμό, διότι όσο εκτελούνται οι διαδικασίες εγγραφής, τα read δεν κάνουν καμία λειτουργία για να διατηρήσουμε την ακεραιότητα των δεδομένων. Συνεπώς ο τρόπος λειτουργίας δεν αλλάζει ιδιαίτερα στην ιδεολογία της.

Ακολουθούν γενικά συμπεράσματα και μια σύντομη ανακεφαλαίωση των υλοποιήσεων.

ΣΤ) ΣΥΜΠΕΡΑΣΜΑΤΑ

Αρχικά για μια σωστή και αποδοτική μηχανική που βασίζεται σε threads και ταυτόχρονες διαδικασίες θα πρεπει να εξασφλίζεται η ακεραιότητα των δεδομένων αλλά και των δομών. Εμείες συγκεκριμένα υλοποιήσαμε τα παρακάτω για την μηχανή:

1. **Δίκαιος διαμοιρασμός των διεργασιών** στα νήματα και επίτευξη στόχου με βάση το ποσοστό αν αυτό δοθεί από τον χρήστη
2. **Δίκαιο διαμοιρασμό και των νημάτων** και με βάση το ποσοστό αν αυτό δοθεί από τον χρήστη
3. **Σωστή καταγραφή στατιστικών** για την σωστή παρατήρηση των υλοποιήσεων με σκοπό την βελτίωση του

Με τα 3 πρώτα , σιγουρεύουμε μια ομοιόμορφη κατανομή που μειώνει τον φόρτο κάθε νήματος με σκοπό την καλύτερη απόδοση και την σταθερότητα τον δομών κατά τις λειτουργίες

4. Αποκλεισμό με σκοπό την ακεραιότητα τον δεδομένων
5. Αποκλεισμό με σκοπό τον παραλληλισμό των αναγνωστών στις διάφορες δομές
6. Συγχρονισμό με σκοπό τον ταυτοχρονισμό μεταξύ των write και read.
7. Ανάλυση στατιστικών για την μελέτη συμπεριφοράς της πολυνηματικής διαδικασίας

Με τα 4 τελευταία διασφαλίζουμε thread-safe engine που σιγουρεύει την και επιβεβαιώνει την σωστή απόδοση του συστήματος, την ακεραιότητα των δομών όταν διεκπεραιώνουν τις λειτουργίες αλλά και την ακεραιότητα των δεδομένων του συστήματος

Σχόλια και ανακεφαλαίωση του αποκλεισμού:

Συμπερασματικά, ο χρόνος που απαιτείται για τις λειτουργίες ανάγνωσης και εγγραφής σε μια δομή δέντρου LSM εξαρτάται από διάφορους παράγοντες, συμπεριλαμβανομένου του αριθμού των νημάτων και του μηχανισμού κλειδώματος που χρησιμοποιείται. Ορισμένα βασικά σημεία που πρέπει να ληφθούν υπόψη είναι τα εξής:

- **Μηχανισμός κλειδώματος**: Η υλοποίηση των κοινών και αποκλειστικών κλειδαριών για τις λειτουργίες ανάγνωσης και εγγραφής, αντίστοιχα, παίζει καθοριστικό ρόλο στον καθορισμό του χρόνου που απαιτείται. Οι κοινόχρηστες κλειδαριές επιτρέπουν ταυτόχρονες λειτουργίες ανάγνωσης, ενώ οι αποκλειστικές κλειδαριές για λειτουργίες εγγραφής μπορεί να προκαλέσουν ανταγωνισμό και χρόνους αναμονής, επηρεάζοντας τη συνολική απόδοση.
- **Αριθμός νημάτων:** Η κατανομή των νημάτων για λειτουργίες ανάγνωσης και εγγραφής επηρεάζει τον απαιτούμενο χρόνο. Καθώς αυξάνεται ο αριθμός των νημάτων, αυξάνεται και ο ανταγωνισμός για κλειδώματα. Ωστόσο, μετά από ένα ορισμένο όριο, η προσθήκη περισσότερων νημάτων μπορεί να μην παρέχει σημαντική βελτίωση του χρόνου μέχρι ένα σημείο, μετά για περεραί.
- **Λειτουργίες εγγραφής:** Οι λειτουργίες εγγραφής σε ένα δέντρο LSM είναι γενικά πιο αργές από τις λειτουργίες ανάγνωσης λόγω της φύσης της δομής δεδομένων. Καθώς αυξάνεται ο αριθμός των νημάτων εγγραφής, αυξάνεται επίσης ο χρόνος αναμονής για την απόκτηση του αποκλειστικού κλειδώματος, οδηγώντας σε μεγαλύτερους χρόνους λειτουργίας εγγραφής.
- **Λειτουργίες ανάγνωσης:** Οι λειτουργίες ανάγνωσης μπορούν να εκτελούνται ταυτόχρονα λόγω του μηχανισμού κοινόχρηστου κλειδώματος. Ωστόσο, η αύξηση του αριθμού των νημάτων ανάγνωσης μπορεί να προκαλέσει μεγαλύτερη αναμονή των νημάτων εγγραφής για την απόκτηση του αποκλειστικού κλειδώματος, επηρεάζοντας τη συνολική ισορροπία μεταξύ των χρόνων ανάγνωσης και εγγραφής.
- **Το υλικό μέρος:** Αυξάνοντας την μνήμη και τους core και παράλληλα το φόρτο που αντέχει το λειτουργικό, μπορείς να βάλεις περισόττερα νήματα και να βελτιώσεις κι άλλο το χρόνο. Όμως ο αριθμός των νημάτων θα πρέπει να είναι αρκετά μεγάλος μετά από ένα σημείο για να είναι εμφανής οι διαφορές

Συνοπτικά, ο χρόνος που απαιτείται για τις λειτουργίες ανάγνωσης και εγγραφής σε ένα δέντρο LSM επηρεάζεται από τον μηχανισμό κλειδώματος, τον αριθμό των νημάτων που διατίθενται σε κάθε λειτουργία και τις εγγενείς διαφορές στην ταχύτητα των λειτουργιών ανάγνωσης και εγγραφής εντός της δομής δεδομένων. Για τη βελτιστοποίηση των επιδόσεων, είναι σημαντικό να εξεταστεί η ισορροπία μεταξύ των νημάτων ανάγνωσης και εγγραφής και να επιλεγεί ένας κατάλληλος μηχανισμός κλειδώματος που ελαχιστοποιεί τον ανταγωνισμό, διασφαλίζοντας παράλληλα τη συνοχή των δεδομένων.