ΜΥΥ 601 ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

1η Εργαστηριακή άσκηση.

ΟΜΑΔΑ:

Μαρούλης Βασίλειος ΑΜ : 4573

Μοτσενίγος Σπυρίδων ΑΜ : 4426

Πρίσκας Σπυρίδων ΑΜ : 4482

Σκοπός: Υλοποίηση πολυνηματικής λειτουργίας των λειτουργιών put και get που περιέχει η μηχανή αποθήκευσης δεδομένων.

Περιεχόμενα:

Αποτελέσματα δοκιμών put():

Αρχικά δοκιμάσαμε τις λειτουργίες read και write με φορτία εργασίας που τις αφορούσαν ώστε να εξοικειωθούμε καλύτερα με την δοθείσα μηχανή αποθήκευσης.

Οπότε αφού δημιουργήσαμε τα εκτελέσιμα με την εντολή make all, δώσαμε πρώτα την εντολή **./kiwi-bench write 50000** η οποία αρχικά εισάγει το πλήθος στοιχείων που δίνουμε ως δεύτερο όρισμα . Η εντολή αυτή καλεί την συνάρτηση db_add.

Η συνάρτηση db_add δέχεται ως παράμετρο ένα ζεύγος κλειδιού-τιμής που πρέπει να προστεθεί στην δομή db_add(db, &sk, &sv);

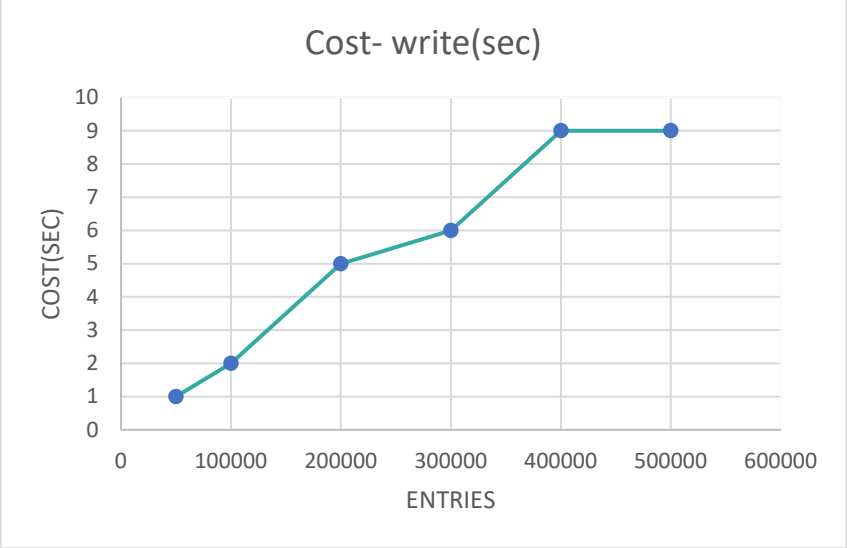
Η ορθότητα της συγκεκριμένης λειτουργίας επιβεβαιώνεται και με τις γραφικές παραστάσεις που ακολουθούν.

Για την αξιόπιστη - ορθή εξαγωγή αποτελεσμάτων μέσω των γραφικών παραστάσεων ακολουθήσαμε την ίδια διαδικασία δίνοντας και κάποιες άλλες τιμές ως ορίσματα, όπως φαίνεται και στις γραφικές παραστάσεις.

Έτσι εκτελέσαμε πειράματα για τις τιμές :

50.000, 100.000, 200.000, 300.000, 400.000, 500.000 writes

Entries	Cost(sec)
50000	1
100000	2
200000	5
300000	6
400000	9
500000	9

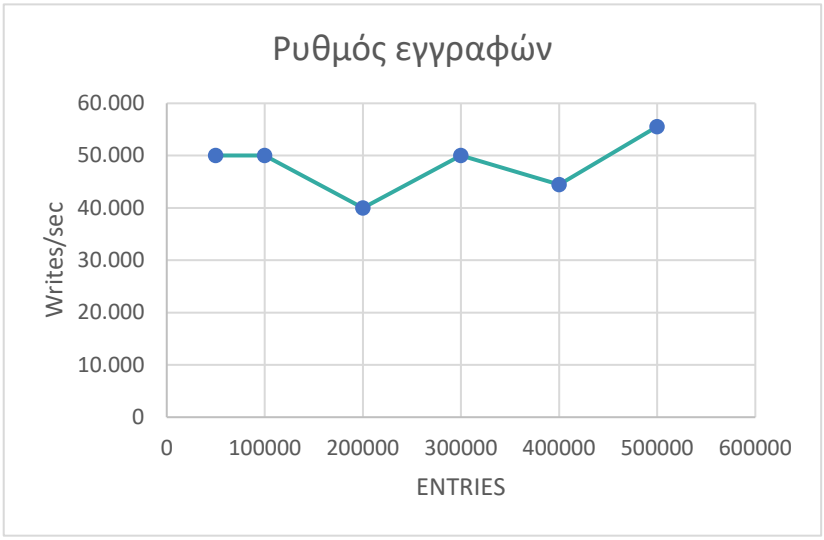


Γραφική παράσταση 1α.

Στην Γραφική παράσταση 1α. Παρατηρούμε τον χρόνο που χρειάζεται ώστε να ολοκληρωθούν οι εγγραφές που ζητούμε κάθε φορά. Όπως ήταν αναμενόμενο ο χρόνος αυξάνεται όσο αυξάνεται το πλήθων των κλειδιών που θέλουμε να εισάγουμε.

Επίσης μπορούμε να δούμε πόσες λειτουργίες γίνονται ανά δευτερόλεπτο.

Entries	Writes/sec
50000	50.000
100000	50.000
200000	40.000
300000	50.000
400000	44.444
500000	55555,6



Στην παραπάνω γραφική παράσταση παρατηρούμε τον ρυθμό με τον οποίο γίνονται οι εγγραφές ανά δευτερόλεπτο σε κάθε περίπτωση που έχουμε ελέγξει.

Merge-compaction:

Αξίζει να σημειωθεί ότι κατά την διαδικασία εισαγωγής κλειδιών παρατηρούμε ότι κατά την εγγραφή γίνονται σωστά και οι λειτουργίες merge και compaction.

Τα πιο πρόσφατα δεδομένα διατηρούνται στην μνήμη στην ταξινομημένη δομή memtable που υλοποιείται από μια δομή Skip list που αποτελείται από πολλές λίστες οργανωμένες σε πολλά επίπεδα ώστε να επιτυγχάνουμε την γρηγορότερη αναζήτηση ενός στοιχείου.

Όταν το memtable γεμίσει μετακινείται στο δίσκο ώστε να αδειάσει η μνήμη και να είναι έτοιμη να δεχτεί νέα στοιχεία. Όσο είναι στο δίσκο υπάρχει ένα αρχείο log που αποθηκεύει προσωρινά τα εισερχόμενα δεδομένα ώστε να μπορούν να είναι δυνατή η ανάκτηση τους σε περίπτωση σφάλματος .

Τα αρχεία που αποθηκεύονται στο δίσκο είναι οργανωμένα σε πολλαπλά επίπεδα των οποίων το μέγεθος αυξάνει με γεωμετρική πρόοδο από τα χαμηλότερα προς τα

υψηλότερα επίπεδα (0..6). Σε ένα αρχείο δίσκου τα δεδομένα είναι ταξινομημένα με βάση τα κλειδιά που περιέχουν συμβολοσειρές.

Έτσι γίνεται merge ώστε τα δεδομένα να μετακινηθούν στο δίσκο και στο αρχείο SSTable και έτσι να αδειάσει χώρος ώστε να προστεθούν νέα δεδομένα.

Όταν εκτελούμε merge τα δεδομένα συγχωνεύονται με τα αρχεία στο επίπεδο 0 ή 1 . Όμως υπάρχει περίπτωση να εισαχθούν παραπάνω δεδομένα στο επίπεδο 0,τότο απαιτείται compaction ώστε τα δεδομένα να συμπτυχθούν , να συγχωνευθούν σε ένα αρχείο και να μετακινηθούν στο επόμενο επίπεδο διαδικασία που θα δημιουργήσει χώρο για την εισαγωγή νέων δεδομένων στην μηχανή αποθήκευσης. Το ίδιο συμβαίνει και στα μεγαλύτερα επίπεδα εάν οι το πλήθος των εγγράφων μας είναι πολύ μεγάλο .

Αυτές οι δύο διαδικασίες γίνονται επιτυχώς στα πειράματα μας όταν δίνουμε μεγάλο πλήθος εγγραφών στην λειτουργία “write” . Όπως φαίνεται και παρακάτω ακολουθείται η διαδικασία που περιγράψαμε.

```
293439 adding key-293439
293440 adding key-293440
293441 adding key-293441
[1356] 15 Mar 12:42:44.853 - sst.c:193 Compactiong due to many files in level 0
293442 adding key-293442
293443 adding key-293443
293444 adding key-293444
[1356] 15 Mar 12:42:44.853 . sst.c:944 Starting compaction. Compaction level: 0 Score: 2.250000
293445 adding key-293445
293446 adding key-293446
293447 adding key-293447
293448 adding key-293448
[1356] 15 Mar 12:42:44.853 - sst.c:825 Extracted range: [key-255254, key-259370]
293449 adding key-293449
[1356] 15 Mar 12:42:44.853 - sst.c:825 Extracted range: [key-255254, key-259370]
293450 adding key-293450
293451 adding key-293451
[1356] 15 Mar 12:42:44.853 . merger.c:28 current FileRange: key-255254 key-259370
293452 adding key-293452
[1356] 15 Mar 12:42:44.853 . merger.c:34        File testdb/si/0/870.sst [key-255254, key-259370]
293453 adding key-293453
293454 adding key-293454
[1356] 15 Mar 12:42:44.853 . merger.c:28 parent FileRange: key-255254 key-259370
293455 adding key-293455
293456 adding key-293456
[1356] 15 Mar 12:42:44.853 - sst.c:825 Extracted range: [key-255254, key-259370]
293457 adding key-293457
293458 adding key-293458
293459 adding key-293459
293460 adding key-293460
[1356] 15 Mar 12:42:44.853 . merger.c:28 final current FileRange: key-255254 key-259370
293461 adding key-293461
293462 adding key-293462
[1356] 15 Mar 12:42:44.853 . merger.c:34        File testdb/si/0/870.sst [key-255254, key-259370]
293463 adding key-293463
293464 adding key-293464
```

```
296424 adding key-296424
[1356] 15 Mar 12:42:44.860 . db.c:52 Starting compaction of the memtable after 4117 insertions and 0 deletions
[1356] 15 Mar 12:42:44.860 . sst.c:595 IN sst_merge the REFCOUNT IS at 2
[1356] 15 Mar 12:42:44.860 - file.c:170 Truncating file testdb/si/71.log to 4195223 bytes
[1356] 15 Mar 12:42:44.860 - sst.c:165 The merge thread received a MERGE job
[1356] 15 Mar 12:42:44.860 . sst.c:166 Merging inside compaction thread
[1356] 15 Mar 12:42:44.860 . sst.c:608 Compacting the memtable to a SST file
[1356] 15 Mar 12:42:44.860 - sst.c:879 Range [key-292307, key-296423] DOES NOT overlap in level 0. Checking others
[1356] 15 Mar 12:42:44.860 - sst.c:825 Extracted range: [key-0, key-999999]
[1356] 15 Mar 12:42:44.860 . sst.c:931 Using level 0 for memtable compaction [key-292307, key-296423]
[1356] 15 Mar 12:42:44.860 . file.c:200 Creating directory structure: testdb/si/0
[1356] 15 Mar 12:42:44.860 . sst.c:633 Compaction of 4117 [4195223 bytes allocated] elements started
[1356] 15 Mar 12:42:44.863 - log.c:159 Log file testdb/si/72.log created
296425 adding key-296425
296426 adding key-296426
```

```
297163 adding key-297163
297164 adding key-297164
297165 adding key-297165
297166 adding key-297166
[1356] 15 Mar 12:42:44.865 . sst.c:170 Merge successfully completed. Releasing the skiplist
297167 adding key-297167
[1356] 15 Mar 12:42:44.865 . skiplist.c:57 SkipList refcount is at 0. Freeing up the structure
297168 adding key-297168
297169 adding key-297169
297170 adding key-297170
297171 adding key-297171
297172 adding key-297172
297173 adding key-297173
```

Αποτελέσματα δοκιμών read():

Τώρα δώσαμε την εντολή **./kiwi-bench read 50000** .Η εντολή αυτή καλεί την συνάρτηση **db_get**.

Η λειτουργία get δέχεται ως παράμετρο ένα κλειδί και αιτείται την αντίστοιχη τιμή εφόσον υπάρχει αποθηκευμένο στην δομή ζεύγος κλειδιού-τιμής με το συγκεκριμένο κλειδί, αλλιώς επιστρέφει σφάλμα αν το κλειδί δεν βρεθεί.

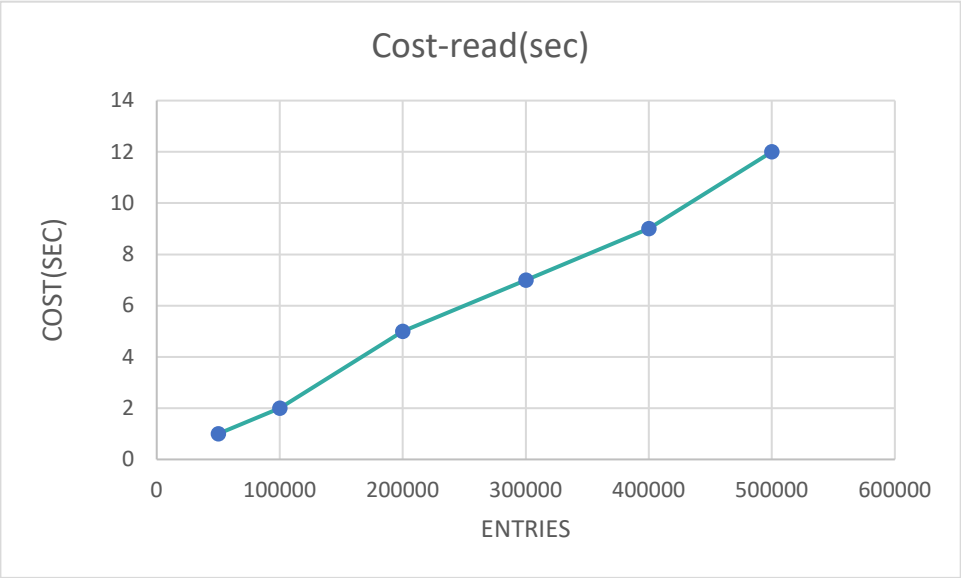
Η ορθότητα της συγκεκριμένης λειτουργίας επιβεβαιώνεται και με τις γραφικές παραστάσεις που ακολουθούν.

Για την αξιόπιστη - ορθή εξαγωγή αποτελεσμάτων μέσω των γραφικών παραστάσεων ακολουθήσαμε την ίδια διαδικασία δίνοντας και κάποιες άλλες τιμές ως ορίσματα, όπως φαίνεται και στις γραφικές παραστάσεις.

Έτσι εκτελέσαμε πειράματα για τις τιμές :

50.000, 100.000, 200.000, 300.000, 400.000, 500.000 reads.

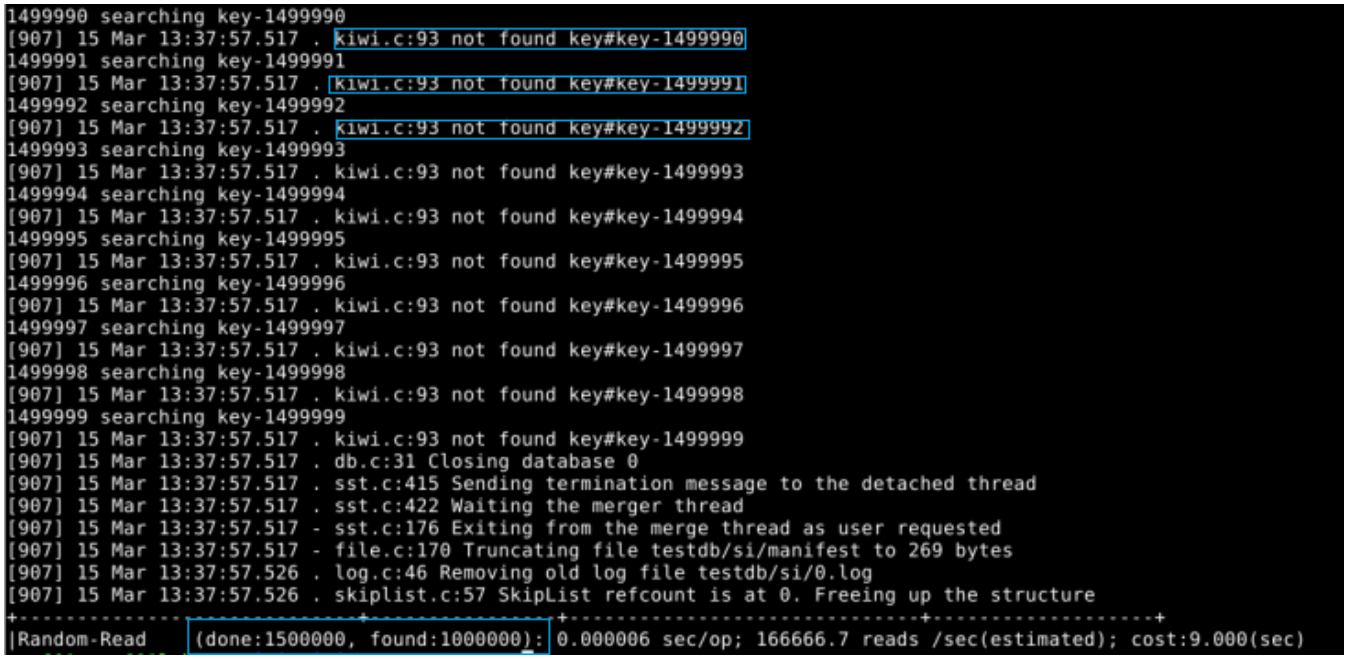
Entries	Cost(sec)
50000	1
100000	2
200000	5
300000	7
400000	9
500000	12



Η λειτουργία db_get αρχικά κάνει αναζήτηση στο Memtable , εάν το κλειδί δεν βρεθεί εκεί προχώρα στα αρχεία sst από το επίπεδο 0 και πηγαίνοντας στα επόμενα εάν αυτό κρίνεται αναγκαίο.

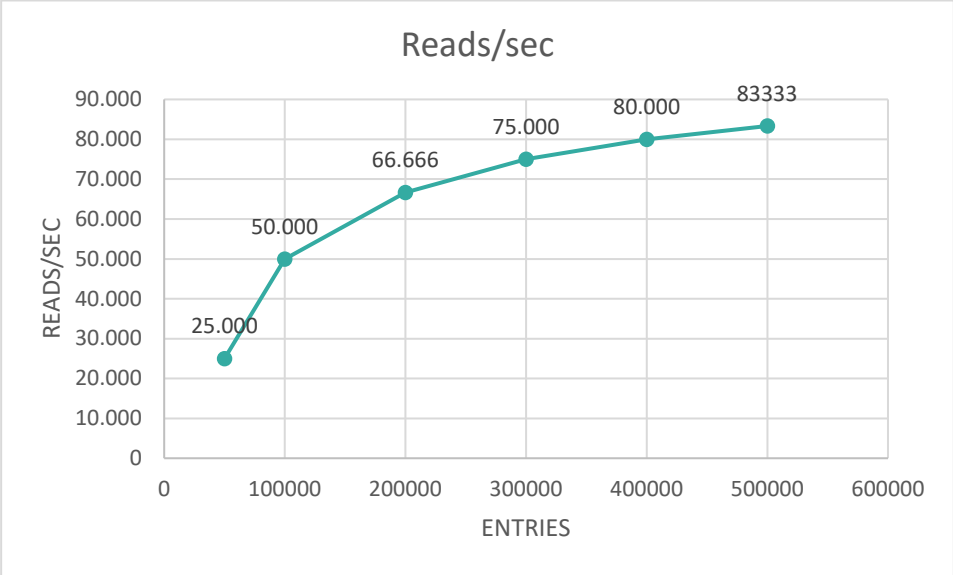
Τα αρχεία sst που επιλέγονται για να γίνει αναζήτηση σε αυτά , επιλέγονται σύμφωνα με το εύρος των κλειδιών που περιέχουν , εάν το κλειδί που θέλουμε να αναζητήσουμε δεν βρίσκεται σε αυτό το εύρος ζώνης βρίσκει άλλο επίπεδο που μπορεί να το περιέχει. Εάν το κλειδί δεν υπάρχει σε κανένα από τα επίπεδα η συνάρτηση

db_get επιστρέφει τιμή 0 και στο terminal εμφανίζεται το μνήμα key not found. Όπως φαίνεται και στην παρακάτω φωτογραφία.



Επίσης βλέπουμε τον ρυθμό με τον οποίο γίνονται οι αναγνώσεις ανά δευτερόλεπτο σε κάθε περίπτωση που έχουμε ελέγξει.

Entries	Reads/sec
50000	25.000
100000	50.000
200000	66.666
300000	75.000
400000	80.000
500000	83.333



Επίσης παρατηρούμε ότι στην αρχή της κλήσης της εντολής ./kiwi-bench readXXXXX γίνεται loading του αρχείων sst και των κατάλληλων των επιπέδων αυτού.

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read 50000
Keys:           16 bytes each
Values:         1000 bytes each
Entries:        50000
IndexSize:      1.2 MB (estimated)
DataSize:       47.9 MB (estimated)
------------------------------------------------------------------------------------------
Date:           Wed Mar 15 13:17:13 2023
CPU:            4 *  AMD Ryzen 5 5600X 6-Core Processor
CPUCache:
[1483] 15 Mar 13:17:13.397 . file.c:200 Creating directory structure: testdb/si
[1483] 15 Mar 13:17:13.397 . file.c:65 Mapping of 2451 bytes for testdb/si/manifest
[1483] 15 Mar 13:17:13.397 . sst.c:331 Loading SST file testdb/si/0/922.sst for level 0 296611 bytes
[1483] 15 Mar 13:17:13.397 . file.c:65 Mapping of 296611 bytes for testdb/si/0/922.sst
[1483] 15 Mar 13:17:13.397 - sst_loader.c:183 Index @ offset: 276807 size: 19732
[1483] 15 Mar 13:17:13.397 - sst_loader.c:184 Meta Block @ offset: 276735 size: 72
[1483] 15 Mar 13:17:13.397 . sst_loader.c:201 Data size:        266843
[1483] 15 Mar 13:17:13.397 . sst_loader.c:203 Index size:       0
[1483] 15 Mar 13:17:13.397 . sst_loader.c:204 Key size:         65872
[1483] 15 Mar 13:17:13.397 . sst_loader.c:205 Num blocks size:  824
[1483] 15 Mar 13:17:13.397 . sst_loader.c:206 Num entries size: 4117
[1483] 15 Mar 13:17:13.397 . sst_loader.c:207 Value size:       4117000
[1483] 15 Mar 13:17:13.397 . sst_loader.c:210 Filter size:      9892
[1483] 15 Mar 13:17:13.397 . sst_loader.c:211 Bloom offset 266843 size: 9892
[1483] 15 Mar 13:17:13.397 . sst.c:342 Smallest key: key-465221 Largest key: key-469337 seeks: 100
[1483] 15 Mar 13:17:13.397 . sst.c:331 Loading SST file testdb/si/0/923.sst for level 0 296224 bytes
[1483] 15 Mar 13:17:13.397 . file.c:65 Mapping of 296224 bytes for testdb/si/0/923.sst
[1483] 15 Mar 13:17:13.397 - sst_loader.c:183 Index @ offset: 276420 size: 19732
[1483] 15 Mar 13:17:13.397 - sst_loader.c:184 Meta Block @ offset: 276348 size: 72
[1483] 15 Mar 13:17:13.397 . sst_loader.c:201 Data size:        266456
[1483] 15 Mar 13:17:13.397 . sst_loader.c:203 Index size:       0
[1483] 15 Mar 13:17:13.397 . sst_loader.c:204 Key size:         65872
[1483] 15 Mar 13:17:13.397 . sst_loader.c:205 Num blocks size:  824
[1483] 15 Mar 13:17:13.397 . sst_loader.c:206 Num entries size: 4117
[1483] 15 Mar 13:17:13.397 . sst_loader.c:207 Value size:       4117000
[1483] 15 Mar 13:17:13.397 . sst_loader.c:210 Filter size:      9892
[1483] 15 Mar 13:17:13.397 . sst_loader.c:211 Bloom offset 266456 size: 9892
[1483] 15 Mar 13:17:13.397 . sst.c:342 Smallest key: key-469338 Largest key: key-473454 seeks: 100
[1483] 15 Mar 13:17:13.397 . sst.c:331 Loading SST file testdb/si/0/924.sst for level 0 294922 bytes
[1483] 15 Mar 13:17:13.397 . file.c:65 Mapping of 294922 bytes for testdb/si/0/924.sst
[1483] 15 Mar 13:17:13.397 - sst_loader.c:183 Index @ offset: 275118 size: 19732
[1483] 15 Mar 13:17:13.397 - sst_loader.c:184 Meta Block @ offset: 275046 size: 72
[1483] 15 Mar 13:17:13.397 . sst_loader.c:201 Data size:        265154
[1483] 15 Mar 13:17:13.397 . sst_loader.c:203 Index size:       0
```

Επέκταση αρχείου bench.c για πολυνηματική λειτουργία .

Στο αρχείο **bench.c** βρίσκουμε τις λειτουργίες "read" , "write" που εκτελέσαμε στα πειράματα που δείξαμε παραπάνω.

Αυτές οι λειτουργίες παίρνονται ως ορίσματα στην main και καλούν τις συναρτήσεις _read_test και _write_test αντίστοιχα. Επίσης το πλήθος των κλειδιών περνιέται σαν όρισμα στη main.

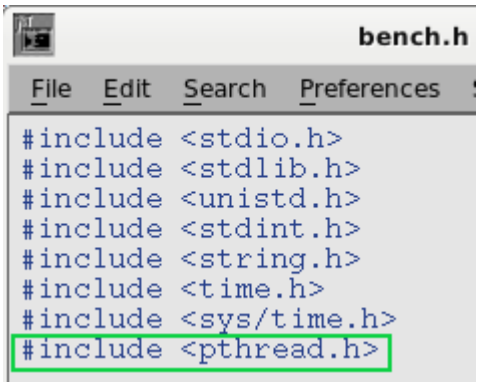
```
int main(int argc,char** argv)
{
        long int count;

        srand(time(NULL));
        if (argc < 3) {
                fprintf(stderr,"Usage: db-bench <write | read> <count>\n");
                exit(1);
        }

        if (strcmp(argv[1], "write") == 0) {
                int r = 0;

                count = atoi(argv[2]);
                _print_header(count);
                _print_environment();
                if (argc == 4)
                        r = 1;
                _write_test(count, r);
        } else if (strcmp(argv[1], "read") == 0) {
                int r = 0;

                count = atoi(argv[2]);
                _print_header(count);
                _print_environment();
                if (argc == 4)
                        r = 1;

                _read_test(count, r);
        } else {
                fprintf(stderr,"Usage: db-bench <write | read> <count> <random>\n");
                exit(1);
        }

        return 1;
}
```

Οι συναρτήσεις _read_test και _write_test είναι υλοποιημένες στο αρχείο kiwi.c , οι οποίες καλούν τις συναρτήσεις db_get , db_add τόσες φορές όσο το πλήθος εισόδου που δίνει ο χρήστης. Αυτό γίνεται όμως για ένα νήμα.

Έτσι εμείς πρέπει να δημιουργήσουμε πολλαπλά νήματα για να γίνει πολυνηματική η μηχανή kiwi.c.

Αρχικά κάναμε import στο αρχείο ‘bench.h’ την βιβλιοθήκη <pthread.h>

Όπως φαίνεται στην διπλανή εικόνα.



Έπειτα στο αρχείο bench.h κάναμε τις κατάλληλες δηλώσεις μεταβλητών.

Όπως για παράδειγμα τα mutex των write και read. Επίσης δηλώσαμε μια δομή την struct data την οποία την χρησιμοποιούμε για να περάσουμε με κατάλληλο τρόπο τις παραμέτρους στις νέες μεθόδους που καλούνται από τα νήματα που δημιουργούμε. Επιπλέον δηλώσαμε μια μεταβλητή int extern found διότι η μεταβλητή αυτή παίρνει τιμή στο kiwi.c και εμείς τη χρησιμοποιούμε στο αρχείο bench.h για να τυπώσουμε το πόσα κλειδιά βρέθηκαν κατά τη λειτουργία ανάγνωσης με πολλαπλά νήματα.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>   //Bibliothiki gia polunimatismo.


#define KSIZE (16)
#define VSIZE (1000)

#define LINE "+-----------------------------+----------------+------------------------------+-------------------+\n"
#define LINE1 "-----------------------------------------------------------------------------------------------------\n"
long long get_ustime_sec(void);
void _random_key(char *key,int length);

pthread_mutex_t write_m , read_m; //dilwsi mutex tou write kai tou read wste na ypologisoume ta
                                 //telika statistika xwris problima otan exoume polla nimata.

pthread_mutex_t tid,tid1; //dilwsi mutex tou write kai tou read

extern int found; //Extren giati allou upoligizetai kai allou tin xrisimopooiw sto bench.c

double totalCostWrite, totalCostRead; //Dilwsi metablhtwn xronou.

struct data {      // orizo mia domi h opoia periexei tis parametrous pou tha perasoume sthn
    long int count;         // To synolo twn leitourgiwn (write - read) pou theloume na ektelesoume
    int r;                  // H metavliti pou dinetai apo tin ekfonisi gia tyxaia kleidia
    int nimata;             // Arithmos twn nimatwn pou theloume na ftiaxoume
};
```

Σε συνέχεια στο αρχείο bench.c αρχικά κάναμε τα κατάλληλα include ώστε να μπορούμε να ανοίξουμε και να κλείσουμε τη βάση μας. Όπως φαίνεται και στη φωτογραφία

```
#include "bench.h"
#include "../engine/db.h"
#define DATAS ("testdb")


DB* db; // dilwsi tou db gia anoigma tis basis.
```

Ύστερα φτιάξαμε 2 νέες συναρτήσεις όπως αυτές που στο τρίτο εργαστήριο στη σελίδα έντεκα. Έτσι οι συναρτήσεις αυτές παίρνουν τους παραμέτρους τους με της βοήθεια της δομής που δηλώσαμε. Οι συναρτήσεις αυτές παίρνουν ως παραμέτρους το σύνολο των λειτουργιών που θα εκτελέσουν, την μεταβλητή r που μας δίνει τη δυνατότητα να έχουμε τυχαία κλειδιά , και τέλος τελευταίο όρισμα παίρνουν τον αριθμό των νημάτων.

Οι συναρτήσεις αυτές ονομάζονται **my_write** και **my_read** . Και καλούνε την συνάρτηση _write_test ή _read_test εκάστοτε.

```
void *my_write(void *arg){    //Sunartisi pou tha ektelesoun ta nimata mas.
                              //Oi parametroi pernane me tin xrisi domis.

    struct data *d = (struct data *) arg; //Orizoume tin domi
    _write_test(d->count, d->r, d->nimata);  //Kaloume tin sunartisi,
                                             //pernontas parametrous apo tin domi

    return 0;
}


void *my_read(void *arg){  //Sunartisi pou tha ektelesoun ta nimata mas.
                           //Oi parametroi pernane me tin xrisi domis.

        struct data *d = (struct data *) arg; //Orizoume tin domi
        _read_test(d->count, d->r, d->nimata); //Kaloume tin sunartisi,
                                               //pernontas parametrous apo tin domi
        return 0;
}
```

Στη συνέχεια έπρεπε να τροποποιήσουμε την main του αρχείου bench.c έτσι ώστε να παίρνει σαν τρίτο όρισμα τον αριθμό των νημάτων. Έτσι όπως φαίνεται και στην **εικόνα 2a** αρχικά δηλώνουμε την μεταβλητή για τον αριθμό των νημάτων στη συνέχεια δηλώνουμε τα νήματα με την εντολή **pthread_t *tid,*tid1;** και δεσμεύουμε δυναμικα μνήμη με βάση τον αριθμό των νημάτων που ζητά ο χρήστης. επίσης αρχικό πιούμε τα mutex με την εντολή **pthread_mutex_init(&write_m,NULL); , pthread_mutex_init(&read_m,NULL);.**

Στη συνέχεια ανοίγουμε τη βάση με την εντολή db = db_open(DATAS);

Επιπλέον ελέγχουμε αν ο χρήστης δώσει λιγότερα από 3 ορίσματα ώστε να το εκτυπώσουμε μήνυμα λάθους.

Έτσι πρώτα ελέγχουμε αν το πρώτο όρισμα είναι η λέξη”write” τότε ο χρήστης θέλει να κάνει εγγραφές και έτσι περνάμε στη δομή το σύνολο των λειτουργιών ,την μεταβλητή r και τον αριθμό των νημάτων.

Η βασικότερη αλλαγή που έγινε είναι αυτή τις επαναλήψεις με τη βοήθεια της for loop ώστε να δημιουργήσουμε τόσα νήματα όσα ζητά ο χρήστης με τη βοήθεια της εντολής **pthread_create(&tid[i],NULL, my_write, (void *)&data1);** έπειτα όσα νήματα δημιούργησα τα περιμένω μέχρι να τελειώσουν με τη βοήθεια της εντολής **pthread_join(tid[i],NULL);.**

```
int main(int argc,char** argv)
{
        long int count;
        int nimata; //Dilwsi metablitis gia ta nimata.

        pthread_t *tid,*tid1; //Dilwsi nimatwn.

        nimata = atoi(argv[3]); //O arithmos twn nimatwn oi 3H parametro
        tid = (pthread_t*) malloc(nimata *sizeof(pthread_t));//Dunamiki deusmeufsi mnimis
        tid1 = (pthread_t*) malloc(nimata *sizeof(pthread_t));


        pthread_mutex_init(&write_m,NULL); //Arxikopoisi tou mutex
        pthread_mutex_init(&read_m,NULL);


        srand(time(NULL));

        db = db_open(DATAS); //Anoigma basis.

        if (argc < 3) {
                fprintf(stderr,"Usage: db-bench <write | read> <count> <nimata>\n");
                exit(1);
        }

        if (strcmp(argv[1], "write") == 0) {

                int r = 0;
                totalCostWrite = 0.0; //metabliti gia ton sunoliko xrono
                count = atoi(argv[2]);
                _print_header(count);
                _print_environment();
                if (argc == 5)
                        r = 1;

                struct data data1 = {count,r,nimata}; //Dinoume times stin domi.

                for( int i = 0; i < nimata; i++){
                        pthread_create(&tid[i],NULL, my_write, (void *)&data1); //Dimiourgia nimatwn.
                }
                for( int i = 0; i < nimata; i++){
                        pthread_join(tid[i],NULL); //Gia kathe nima pou ftiaxnw perimenw mexri na
                                                   //teleiwsoun ola.
                }

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

Εικόνα 2a.

Το ίδιο κάνουμε και για τη περίπτωση που ο χρήστης θέλει να διαβάσει κάποιο αριθμό κλειδιών με πολλαπλά νήματα. Όπως φαίνεται στην **εικόνα 2b.**

```
else if (strcmp(argv[1], "read") == 0) {
        int r = 0;
        totalCostRead = 0.0; //metabliti gia ton sunoliko xrono
        count = atoi(argv[2]);
        _print_header(count);
        _print_environment();
        if (argc == 5)
                r = 1;
        struct data data2 = {count,r,nimata}; //Dinoume times stin domi.
        for( int i = 0; i < nimata; i++){
                pthread_create(&tid1[i],NULL, my_read, (void *)&data2); //Dimiourgia nimatwn.
        }
        for( int i = 0; i < nimata; i++){
                pthread_join(tid1[i],NULL); //Gia kathe nima pou ftiaxnw perimenw mexri na,
                                            //teleiwsoun ola.
        }

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

Εικόνα 2b.

Επιπλέον στο τέλος αποδεσμεύουμε τη μνήμη που είχαμε δέσμευση στην αρχή του bench.

```
        else {
                fprintf(stderr,"Usage: db-bench <write | read> <count> <random>\n");
                exit(1);
        }
        free(tid); //Apodeusmeusi
        free(tid1);
        db_close(db); //Kleinw tin basi.

        return 1;
}
```

Αλλαγές χρειάστηκαν και στο αρχείο kiwi.c. Αρχικά αρχικοποιήσαμε στην αρχή του αρχείου τη μεταβλητή found που προηγουμένως είχαμε δήλωση στο αρχείο bench.h.

```
#include <string.h>
#include "../engine/db.h"
#include "../engine/variant.h"
#include "bench.h"
#define DATAS ("testdb")

int  found =0 ; //arxikopoisi tou found , dhlwmeno sto bench.h
DB *db;
```

Στη συνέχεια επεκτείναμε τους παραμέτρους των συναρτήσεων _write_test και _read_test αφού προσθέσαμε τον αριθμό των νημάτων ως παράμετρο.

Έτσι στη συνάρτηση έχουμε ορίσει μια μεταβλητή int x έτσι ώστε σε αυτή να μοιράσουμε ισότιμα τις λειτουργίες για κάθε νήμα. τέλος κλειδώνουμε το mutex που έχουμε φτιάξει γιατί εκάστοτε συνάρτηση ώστε να τυπώσουμε σωστά το τελικό αποτέλεσμα όσον αφορά το χρόνο που έκανε να ολοκληρωθεί η λειτουργία που ζήτησε ο χρήστης έτσι προσθέτουμε στο συνολικό κόστος το χρόνο που παίρνει το κάθε νήμα να ολοκληρωθεί. Όπως φαίνεται και στην **εικόνα 2c.**

```
void _read_test(long int count, int r, int nimata)
{
        int i,x;
        int ret;
        double cost;
        long long start,end;
        Variant sk, sv;

        char key[KSIZE + 1];

        x = count / nimata; //Moirazw isotima tiw leitourgies se katse nima.

        start = get_ustime_sec();

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

                /* if you want to test random write, use the following */
                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) {
                        //db_free_data(sv.mem);
                        found++;
                } else {
                        INFO("not found key#%s",
                                        sk.mem);
        }

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

        pthread_mutex_lock(&read_m); //kleidwnw to mutex
        totalCostRead = totalCostRead + cost;//Prosthetw sto sunoliko kostos to xronou ektelis
                                             //tou kathe nimatos gia na kratisw to apotelesma.
        pthread_mutex_unlock(&read_m);  //xekleidwnw to  mutex
        printf(LINE);
}
```

Εικόνα 2c.

Το ίδιο κάναμε και για την συνάρτηση _write_test. Όπως φαίνεται και στην **εικόνα 2d.**

```
void _write_test(long int count, int r,int nimata)
{
        int i ,x;
        double cost;
        long long start,end;
        Variant sk, sv;

        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);

        x = count / nimata; //Moirazw isotima tiw leitourgies se katse nima.

        start = get_ustime_sec();
        for (i = 0; i < x; i++) {
                if (r==1)
                        _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);
                }
        }
        end = get_ustime_sec();
        cost = end -start;

        pthread_mutex_lock(&write_m); //kleidwnw to mutex
        totalCostWrite = totalCostWrite + cost; //Prosthetw sto sunoliko kostos to xronou ektelis
                                                //tou kathe nimatos gia na kratisw to apotelesma.
        pthread_mutex_unlock(&write_m); //xekleidwnw to  mutex
        printf(LINE);
}
```

Εικόνα 2d.

Έτσι για να τρέξει κανείς το bench θα πρέπει να πληκτρολογήσει στο τερματικό **./kiwi-bench** '1η παράμετρος ' '2η παράμετρος ' '3η παράμετρος '.

- **1η παράμετρος:** Η **λειτουργία** που θέλει να εκτελέσει ο χρήστης.(read,write, writeread)

- **2η παράμετρος:** Το **πλήθος** των στοιχείων που θέλει να γράψει ή να διαβάσει ο χρήστης.

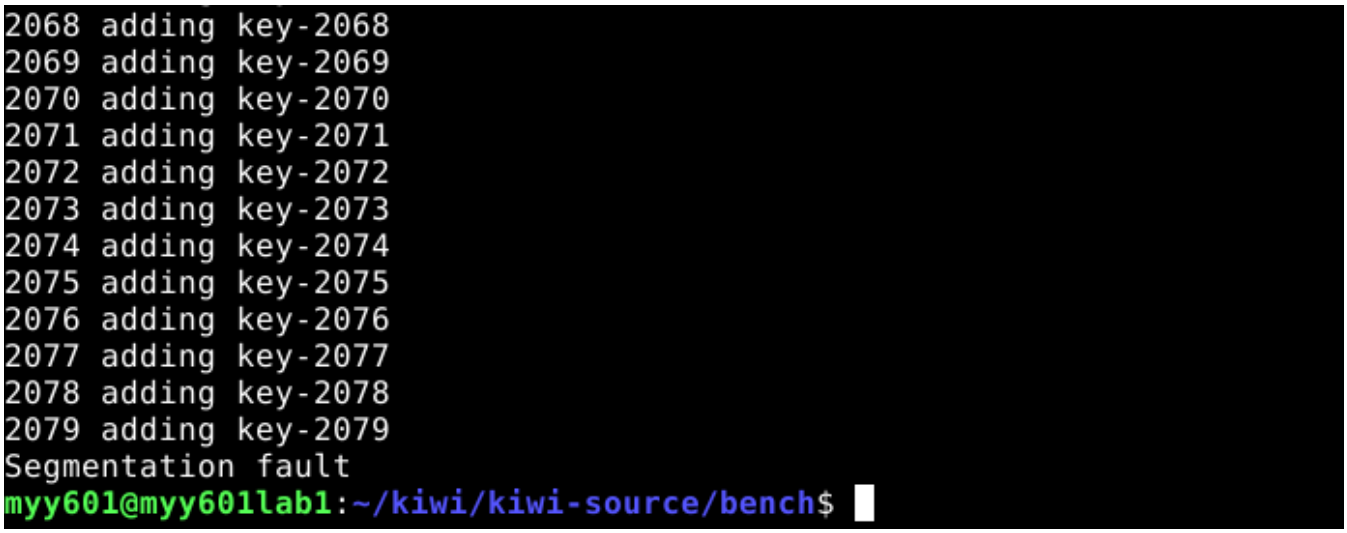
- **3η παράμετρος:** Ο **αριθμός** των **νημάτων**.

Αποτελέσματα πολυνηματικής λειτουργίας.

Αφού ολοκληρώσαμε την κατασκευή της πολυνηματικής λειτουργίας τρέξαμε πειράματα με είσοδο τις 300.000 λειτουργίες τόσο για read όσο και για write.

Write:

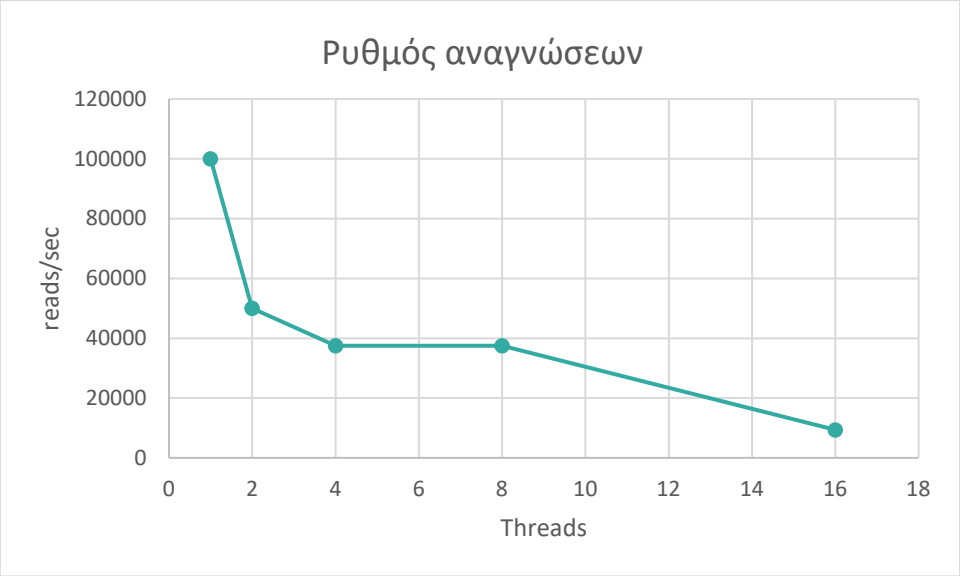
Όπως περιμέναμε η λειτουργία αυτή δεν είναι εφικτό να τερματίσει με πολλαπλά νήματα αφού οι εγγραφείς. Αφού κατά τη διαδικασία της εγγραφής κάποιου κλειδιού στη βάση έρθει και μια ακόμη εγγραφή στη βάση από κάποιο άλλο νήμα τότε θα δημιουργηθεί αστάθεια στο σύστημά μας. έτσι όπως το περιμέναμε η διαδικασία δεν τερματίζει ποτέ και τυπώνει μήνυμα λάθους segmentation fault.



Read:

Δεν συμβαίνει το ίδιο για τη λειτουργία της ανάγνωσης και έτσι έχουμε τρέξει πειράματα για **300.000 εισόδους** με πολλαπλά νήματα τα αποτελέσματα των πειραμάτων αυτών είναι τα αναμενόμενα.

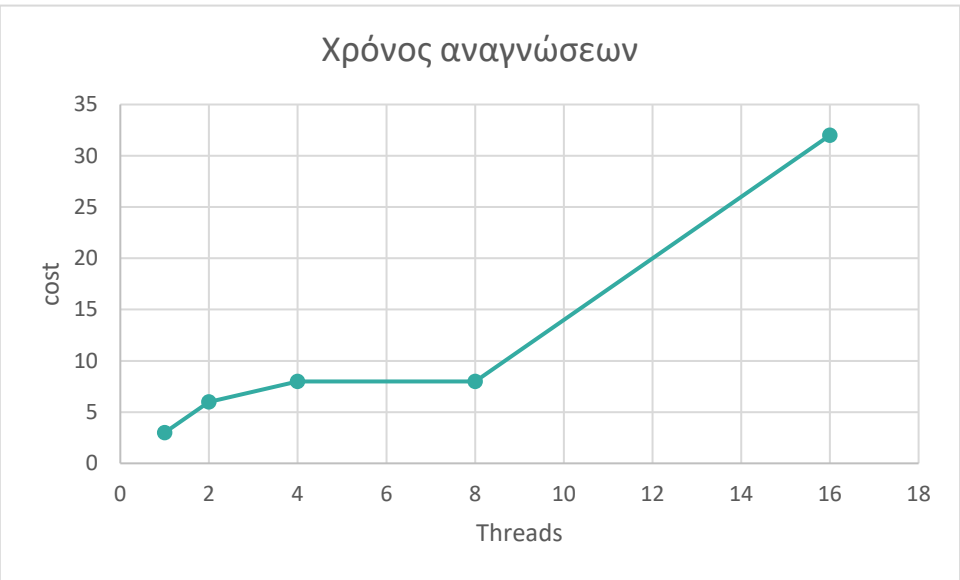
Threads	reads/sec
1	100000
2	50000
4	37500
8	37500
16	9375



Έτσι όπως φαίνεται και στο παραπάνω γράφημα η αύξηση των threads δεν ισοδυναμεί με επιτάχυνση της της λειτουργίας.

Τον αριθμό των νημάτων τον επιλέξαμε με βάση τη δύναμη του 2

Threads	cost
1	3
2	6
4	8
8	8
16	32



Επέκταση του bench για ‘writeread’ λειτουργία.

Τέλος επεκτείναμε το αρχείο bench.c ώστε να μπορεί ο χρήστης να εκτελεί λειτουργία εγγραφής και ανάγνωσης ταυτόχρονα και να μπορεί να δίνει το επιθυμητό ποσοστό λειτουργιών . για αυτό δηλώσαμε ένα ακόμη μήνυμα στο αρχείο bench.h και το αρχικοποιήσαμε στο αρχείο bench.c.

Η υλοποίηση της τελευταίας επιλογής writeread είναι παρόμοια με αυτές των απλών write , read. Με μόνη διαφορά ότι στη δημιουργία των νημάτων δημιουργούμε τον ίδιο αριθμό νημάτων που θα εκτελέσουν εγγραφές αλλά και αναγνώσεις. Αν το ποσοστό που δίνει ο χρήστης είναι άνω των 50% τότε θα γίνουν περισσότερες εγγραφές αφού αν συμβεί το αντίθετο τότε πολύ απλά οι αναγνώσεις δεν θα βρίσκουν το κλειδί.

Η υλοποίηση αυτή φαίνεται στις παρακάτω εικόνες.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <string.h>
#include <time.h>
#include <sys/time.h>
#include <pthread.h>   //Bibliothiki gia polunimatismo.


#define KSIZE (16)
#define VSIZE (1000)

#define LINE "+-----------------------------+----------------+------------------------------+-------------------+\n"
#define LINE1 "-----------------------------------------------------------------------------------------------------\n"
long long get_ustime_sec(void);
void _random_key(char *key,int length);


pthread_mutex_t write_m , read_m; //dilwsi mutex tou write kai tou read wste na ypologisoume ta
                                 //telika statistika xwris problima otan exoume polla nimata.

pthread_mutex_t tid,tid1,tid2,tid3; //dilwsi mutex tou write kai tou read kai tou writeread

extern int found; //Extren giati allou upoligizetai kai allou tin xrisimopooiw sto bench.c

double totalCostWrite, totalCostRead; //Dilwsi metablhtwn xronou.

struct data {      // orizo mia domi h opoia periexei tis parametrous pou tha perasoume sthn
    long int count;         // To synolo twn leitourgiwn (write - read) pou theloume na ektelesoume
    int r;                  // H metavliti pou dinetai apo tin ekfonisi gia tyxaia kleidia
    int nimata;             // Arithmos twn nimatwn pou theloume na ftiaxoume
};
```

```
else if(strcmp(argv[1], "writeread") == 0){
        int r = 0;
        int count1 =0;
        int count2 = 0; //Dilwsi metablhtwn gia na ypolgisw tis leitoyrgies.

        if(argc < 5){
                fprintf(stderr,"Usage: db-bench <writeread> <count> <threads> <pososto> <random>\n");
                exit(1);
        }
        int pososto = atoi(argv[4]);
        db = db_open(DATAS); //anoigw tin basi

        count1 =  (atoi(argv[2])*(pososto*0.01));
        count2 =  (atoi(argv[2])*(1 - pososto*0.01));


        _print_environment();
        if (argc == 6)
                r = 1;

        struct data data3 = {count1,r,nimata}; //Dinoume times stin domi.
        struct data data4 = {count2,r,nimata}; //Dinoume times stin domi.
        for( int i = 0; i < nimata; i++){

                //Dimiourgia nimatwn pou tha kalesoun tin my_write.
                pthread_create(&tid2[i],NULL, my_write, (void *)&data3);

                //Dimiourgia nimatwn pou tha kalesoun tin my_read.
                pthread_create(&tid3[i],NULL, my_read, (void *)&data4);
        }
        for( int i = 0; i < nimata; i++){
                pthread_join(tid2[i],NULL); //Gia kathe nima pou ftiaxnw perimenw mexri na,
                                            //teleiwsoun ola.

                pthread_join(tid3[i],NULL); //Gia kathe nima pou ftiaxnw perimenw na teleiwsoun.
        }

        // emfanizo ta apotelesmata gia count1 write leitourgies
        printf("|Random-Write   (done:%d): %.6f sec/op; %.1f writes/sec(estimated); cost:%.3f(sec);\n"
        ,count1, (double)(totalCostWrite / count1)
        ,(double)(count1 / totalCostWrite),totalCostWrite);


        // emfanizo ta apotelesmata gia count2 read leitourgies
        printf("|Random-Read    (done:%d, found:%d): %.6f sec/op; %.1f reads /sec(estimated); cost:%.3f(sec)\n",
        count2, found,
        (double)(totalCostRead / count2),
        (double)(count2 / totalCostRead),
        totalCostRead);
}
```

Τέλος αποδεσμεύουμε τη μνήμη που είχαμε αρχικά δεσμεύσει.

```
        free(tid); //Apodeusmeusi
        free(tid1);
        free(tid2);
        free(tid3);
        db_close(db); //Kleinw tin basi.

        return 1;
}
```

Έτσι για να τρέξει κανείς το bench θα πρέπει να πληκτρολογήσει στο τερματικό **./kiwi-bench** '1η παράμετρος ' '2η παράμετρος ' '3η παράμετρος ' '4η παράμετρος'.

- **1η παράμετρος:** Η **λειτουργία** που θέλει να εκτελέσει ο χρήστης.(**read,write, writeread**)

- **2η παράμετρος:** Το **πλήθος** των στοιχείων που θέλει να γράψει ή να διαβάσει ο χρήστης.

- **3η παράμετρος:** Ο **αριθμός** των **νημάτων**.

- **4η παράμετρος**: Το **ποσοστό**

Αποτελέσματα writeread.

Στη συνέχεια τρέξαμε πειράματα με είσοδο τις 300.000 λειτουργίες και διαφορετικά ποσοστά και έτσι παρατηρήσαμε τη σωστή λειτουργία της writeread. Όπως αναφέραμε αν ο χρήστης δώσει ποσοστό μεγαλύτερο του 50% τότε θα εκτελεστούν περισσότερες εγγραφές ενώ αν δώσει ποσοστό μικρότερο του 50% θα συμβούν περισσότερες αναγνώσεις.

Τα πειράματα εκτελέστηκαν για ένα νήμα αφού για 2 ή περισσότερα χωρίς να έχουμε δημιουργήσει την καθολική κλειδαριά κατά τη λειτουργία της εγγραφής έχουμε το αναμενόμενο αποτέλεσμα δηλαδή segmention fault.

./kiwi-bench writeread 300000 1 50

```
+-----------------------------+---------------+-----------------------------+-------------------+
|Random-Write   (done:150000): 0.000027 sec/op; 37500.0 writes/sec(estimated); cost:4.000(sec);
|Random-Read    (done:150000, found:150000): 0.000033 sec/op; 30000.0 reads /sec(estimated); cost:5.000(sec)
```

./kiwi-bench writeread 300000 1 80

```
+-----------------------------+---------------+-----------------------------+-------------------+
|Random-Write   (done:240000): 0.000025 sec/op; 40000.0 writes/sec(estimated); cost:6.000(sec);
|Random-Read    (done:59999, found:59999): 0.000033 sec/op; 29999.5 reads /sec(estimated); cost:2.000(sec)
```

./kiwi-bench writeread 300000 1 25

```
+-----------------------------+---------------+-----------------------------+-------------------+
|Random-Write   (done:75000): 0.000013 sec/op; 75000.0 writes/sec(estimated); cost:1.000(sec);
|Random-Read    (done:225000, found:62814): 0.000027 sec/op; 37500.0 reads /sec(estimated); cost:6.000(sec)
```

Εδώ παρατηρούμε ότι γίνονται περισσότερες αναγνώσεις.

Επίσης αξίζει να αναφέρουμε ότι κατά τη διαδικασία γίνονται τόσο οι εγγραφές όσο και αναγνώσεις έτσι όπως φαίνεται στην παρακάτω εικόνα.

```
12131 searching key-12131
12132 searching key-12132
12133 searching key-12133
12134 searching key-12134
19817 adding key-19817
19818 adding key-19818
19819 adding key-19819
19820 adding key-19820
12135 searching key-12135
12136 searching key-12136
12137 searching key-12137
12138 searching key-12138
12139 searching key-12139
19821 adding key-19821
19822 adding key-19822
19823 adding key-19823
```

Δημιουργία καθολικής κλειδαριάς.

Στη συνέχεια κατασκευάσαμε πιο απλή προσέγγιση ώστε να τρέχει μόνο μία λειτουργία την φορά κάτι που εξασφαλίζεται με αμοιβαίο αποκλεισμό.

Σε αυτή την φάση σκεφτήκαμε να υλοποιήσουμε μία καθολική κλειδαριά.

Ετσι μεσα στο αρχειο db.h και συγκεκριμενα στην δομη db αρχικοιποιησαμε το mutex **pthread_mutex_init(&self->my_mutex, NULL);**

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

    pthread_mutex_t my_mutex; //Dhlwnw ena mutex tis domis

} DB;
```

επίσης στο αρχείο db.c και συγκεκριμένα στην μέθοδο dp_open_ex κάναμε την δήλωση αυτού του mutex με την εντολή **pthread_mutex_init(&self->my_mutex,NULL);**

```
DB* db_open_ex(const char* basedir, uint64_t cache_size)
{
    DB* self = calloc(1, sizeof(DB));

    pthread_mutex_init(&self->my_mutex,NULL); //Arxikopoisi tou mutex
    if (!self)
        PANIC("NULL allocation");

    strncpy(self->basedir, basedir, MAX_FILENAME);
    self->sst = sst_new(basedir, cache_size);

    Log* log = log_new(self->sst->basedir);
    self->memtable = memtable_new(log);

    return self;
}
```

Για την το κλείδωμα και το ξεκλείδωμα αυτού του mutex είναι υπεύθυνες οι παρακάτω εντολές.

Pthread_mutex_lock(&self->my_mutex);

pthread_mutex_unlock(&self->my_mutex);

Με την εντολή **Pthread_mutex_lock(&self->my_mutex);** κλειδώνουμε τις κρίσιμες περιοχές στις συναρτήσεις db_add και db_get ώστε όταν κληθεί μία από τις δύο συναρτήσεις τότε να κλειδώνει το mutex ,με συνέπεια αν η άλλη συνάρτηση κληθεί και επιδιώξει να εισέλθει στην δική της κρίσιμη περιοχή να δει ότι το mutex είναι κλειδωμένο και έτσι να περιμένει ώστε να ολοκληρωθεί η άλλη συνάρτηση. Όταν η συνάρτηση που κλήθηκε πρώτη ολοκληρώσει τότε πρέπει να ξεκλειδώσει την κλειδαριά με την εντολή **pthread_mutex_unlock(&self->my_mutex);**

Ενώ η δεύτερη εντολή που κλήθηκε με την σειρά της, να κλειδώσει και να εκτελέσει τον κώδικα που υπάρχει στην δική της κρίσιμη περιοχή.

Πάμε τώρα στην δική μας περίπτωση , πρέπει να κλειδώσουμε την κρίσιμη περιοχή των συναρτήσεων db_add και db_get.
Αρχικά θα μιλήσουμε για την συνάρτηση **db_add.**

Στην συνάρτηση αρχικοποιούμε μια μεταβλητή int add_x , αυτή η μεταβλητή είναι ικανή και αναγκαία συνθήκη για την σωστή λειτουργία της συνάρτησης και για το κλείδωμα της κρίσιμης περιοχής της . Η μεταβλητή αυτή ορίζεται ως add_x = memtable_add(self->memtable, key, value); Το δεξιά μέλος αυτής της εξίσωσης είναι ότι μου επέστρεφε η συνάρτηση db_add πριν την δική μας παρέμβαση. Επειδή όμως η memtable_add(self->memtable, key, value); αποτελεί κρίσιμη περιοχή της συνάρτησης έπρεπε να κρατήσουμε την τιμή που επιστρέφει και μετρά να ξεκλειδώσουμε και για αυτό χρησιμοποιούμε την μεταβλητή add_x. Η μεταβλητή add_x παίρνει την τιμή 1

όταν μια εγγραφή πραγματοποιηθεί σωστά ενώ παίρνει την τιμή 0 όταν η εγγραφεί δεν ολοκληρωθεί σωστά . Μετά κλειδώνουμε την κρίσιμη περιοχή της συνάρτησης , μέχρι να φτάσουμε στο τέλος και να την ξεκλειδώσουμε. Τέλος επιστρέφουμε την τιμή του add_x.

```
int db_add(DB* self, Variant* key, Variant* value)
{
    int add_x; //metavliti pou xreiazomaste gia tin sosti leitourgia ths db_add
    pthread_mutex_lock(&self->my_mutex); //Kleidwma my_mutex sti krisimi perioxi

    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);
    }
    add_x = memtable_add(self->memtable, key, value); //An to kleidi moy eggrafei swsta pairnei tin
                                                      //timh 1 alliws tn timi 0.
    pthread_mutex_unlock(&self->my_mutex); //xekleidwma my_mutex otan bgainw apo krisimi perioxi

    return add_x; //epistrefw to add_x
}
```

Για την db_get , πάλι αρχικοποιούμε μια μεταβλητή int get_x που αποτελεί αναγκαία συνθήκη για την σωστή λειτουργία της συνάρτησης όπως για την db_add.

Η μεταβλητή αυτή ορίζεται ως get_x = sst_get(self->sst, key, value); Το δεξιά μέλος αυτής της εξίσωσης είναι η τιμή που επέστρεφε η συνάρτηση gb_get χωρίς την δική μας παρέμβαση.
Και σε αυτή τη περίπτωση η μεταβλητή get_x παίρνει την τιμή 1 αν το κλειδί που ζητάμε βρέθηκε.

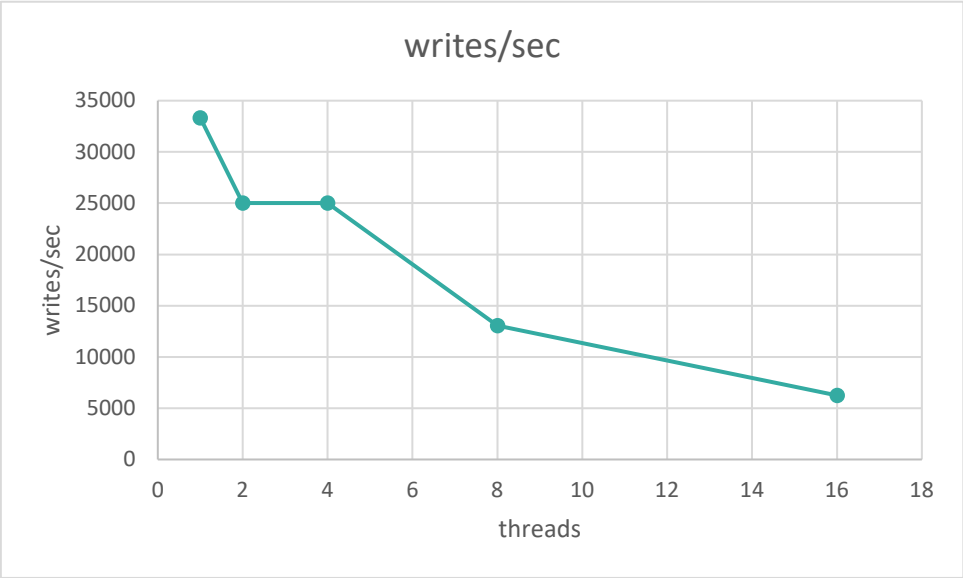
```
int db_get(DB* self, Variant* key, Variant* value)
{
    int get_x; //metavliti pou xreiazomaste gia tin sosti leitourgia ths db_get

    pthread_mutex_lock(&self->my_mutex); //Kleidwma my_mutex sti krisimi perioxi

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

        get_x = sst_get(self->sst, key, value);

    }
    pthread_mutex_unlock(&self->my_mutex); //xekleidwma my_mutex otan bgainw apo krisimi perioxi

    return get_x; //epistrefoume ti metabliti get_x
}
```

Μετά κλειδώνουμε την κρίσιμη περιοχή της συνάρτησης , μέχρι να φτάσουμε στο τέλος και να την ξεκλειδώσουμε. Τέλος επιστρέφουμε την τιμή του get_x.

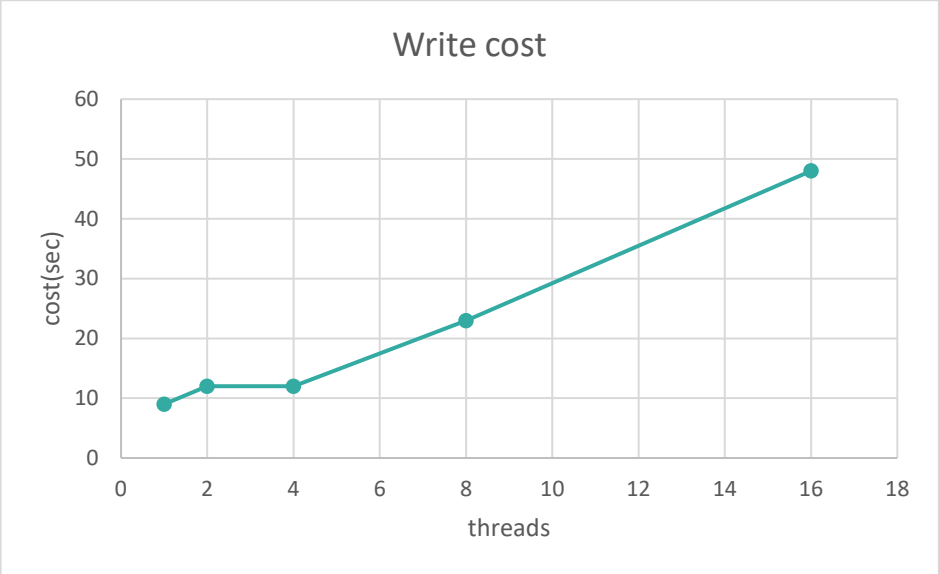
Πειράματα-αποτελέσματα με την ύπαρξη της καθολικής κλειδαριάς.

Αρχικά θα ξεκινήσουμε με τη **λειτουργία εγγραφής** διότι όπως είδαμε παραπάνω με την ύπαρξη πολλαπλών νημάτων χωρίς κάποιο είδος ταυτοχρονισμού η λειτουργία δεν τερμάτιζε ποτέ όπως ήταν αναμενόμενο. Τώρα με την ύπαρξη της καθολικής κλειδαριάς η λειτουργία εγγραφής δουλεύει σωστά και έτσι έχουμε τρέξει **πειράματα για 300.000 εισόδους** και για πολλαπλά νήματα.

threads	writes/sec
1	33333
2	25000
4	25000
8	13043
16	6250



threads	cost
1	9
2	12
4	12
8	23
16	48

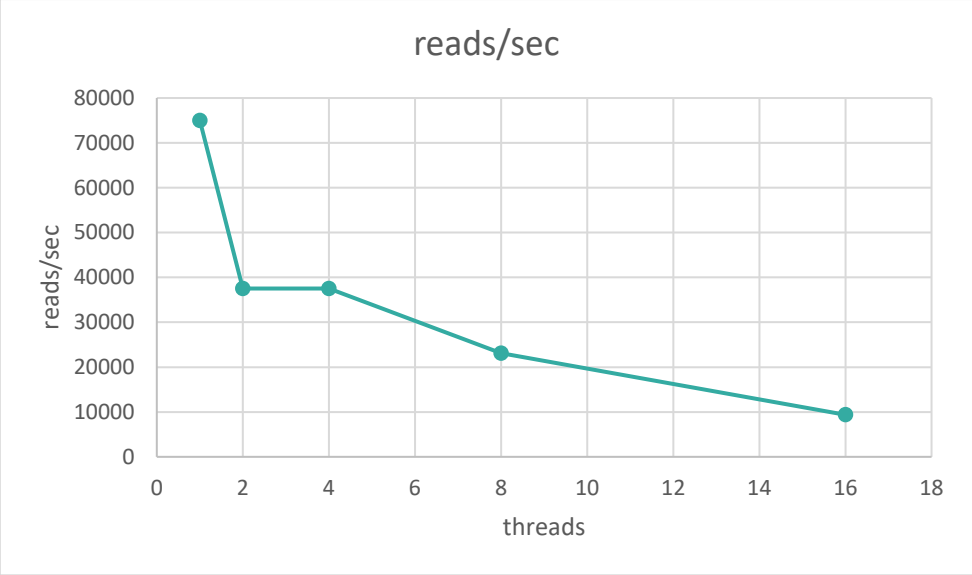


Όπως είναι αναμενόμενο η αύξηση των νημάτων επιφέρει αύξηση στο κόστος δηλαδή στην διάρκεια πού γίνονται οι εγγραφές αυτό συμβαίνει διότι πολλαπλά νήματα

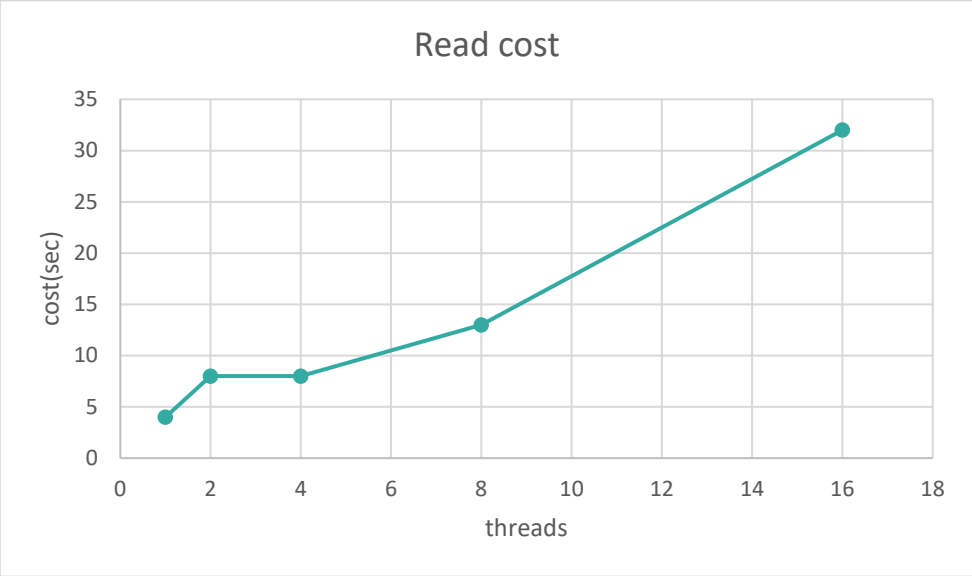
προσπαθούν να γράψουν στη βάση και έτσι η βάση κλειδώνει περισσότερες φορές με αποτέλεσμα την αύξηση των καθυστερήσεων.

Στη συνέχεια εκτελέσαμε πειράματα για την λειτουργία ανάγνωσης όπως κάναμε και στην αρχή όπου υλοποιήσαμε την πολυνηματική λειτουργία. Ξανά, το πλήθος των αναγνώσεων είναι 300.000 και κάναμε πειράματα για πολλαπλά νήματα με την ύπαρξη της καθολικής κλειδαριάς μας.

threads	reads/sec
1	75000
2	37500
4	37500
8	23076
16	9375



threads	cost
1	4
2	8
4	8
8	13
16	32



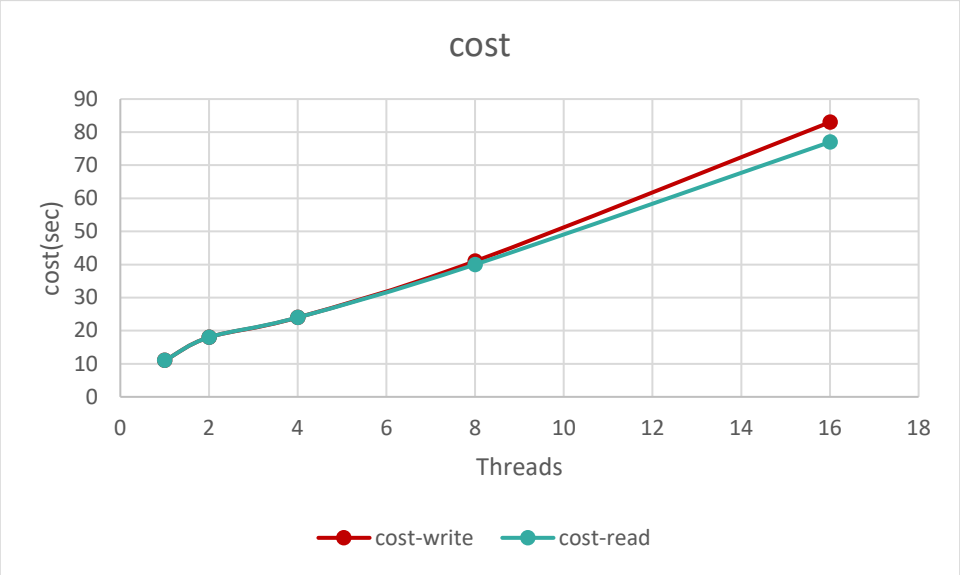
Η λειτουργία read τερμάτιζε σωστά και πριν, δηλαδή χωρίς την ύπαρξη της καθολικής κλειδαριάς. Έτσι τα αποτελέσματα είναι τα αναμενόμενα , δηλαδή ίδια με πριν.

Επιπλέον κάναμε πειράματα για τη λειτουργία writeread με 600.000 εισόδους και 50% ποσοστό ώστε να κάνει 300.000 εγγραφές και 300.000 αναγνώσεις. Και σε αυτή την περίπτωση μπορούμε να πάρουμε τα στατιστικά αφού έχουμε φτιάξει την καθολική κλειδαριά διότι πριν η λειτουργία εγγραφής δεν τερματίζεται ποτέ όπως αναφέρθηκε και παραπάνω.

threads	writes/sec	reads/sec
1	27272	27272
2	16666	16666
4	12500	12500
8	7317	7500
16	3614	3896

Όπως παρατηρούμε ότι γίνονται ο ίδιος λειτουργιών το δευτερόλεπτο αυτό είναι λογικό αφού εκτελούν το ίδιο πλήθος λειτουργιών δηλαδή 300.000.

threads	cost-write	cost-read
1	11	11
2	18	18
4	24	24
8	41	40
16	83	77



Όπως φαίνεται και στο παραπάνω γράφημα το κόστος και των 2 λειτουργιών είναι σχεδόν το ίδιο αυτό είναι αναμενόμενο αφού εκτελούν το ίδιο πλήθος λειτουργιών και η καθολική κλειδαριά δεν δίνει προτεραιότητα σε κάποια από τους από τις 2 λειτουργίες. Βλέπουμε μια μικρή διαφορά ανάμεσα στις 2 αυτό συμβαίνει για τη λειτουργία εγγραφής είναι γενικότερα πιο χρονοβόρα από τη λειτουργία της ανάγνωσης.

Όπως περιμέναμε τα αποτελέσματα δεν διαφέρουν πάρα πολύ με ή χωρίς την ύπαρξη της καθολικής κλειδαριάς αυτό οφείλεται στο γεγονός ότι μια τέτοια καθολική κλειδαριά δεν παρέχει πολύ καλό επίπεδο ταυτοχρονισμού.

Πρόβλημα αναγνωστέων - γραφών:

Στη συνέχεια δώσαμε μια λύση στο πρόβλημα αναγνώστη των γραφών με προτεραιότητα στους γραφείς έτσι ώστε να πετύχουμε καλύτερο συγχρονισμό από ότι αυτόν που μας έδωσε η καθολική κλειδαριά. Τον αλγόριθμο που χρησιμοποιήσαμε τον βρήκαμε στο διαδίκτυο και τον τροποποιήσαμε κατάλληλα ώστε να τον φέρουμε στο δικό μας πρόβλημα.

Αρχικά ορίσαμε μια νέα μεταβλητή συνθήκης στο αρχείο db.h τώρα και 3 ακόμη μεταβλητές ακέραιου τύπου. Αυτές οι μεταβλητές όπως φαίνονται και στα σχόλια του κώδικα κρατάνε τον αριθμό αυτών που θέλουν να γράψουν στην βάση, αυτών που διαβάζουν κάποια συγκεκριμένη στιγμή στη βάση και τέλος όσο με γράφουν κάποια συγκεκριμένη στιγμή στη βάση αυτές οι μεταβλητές Παίζουν καθοριστικό ρόλο ώστε να μπορούμε να ελέγχουμε οποιαδήποτε στιγμή εάν κάποιος γράφει στη βάση εάν κάποιος διαβάζει από τη βάση ή κάποιος εγγραφής θέλει να γράψει στη βάση.

```
typedef struct _db {
//    char basedir[MAX_FILENAME];
    char basedir[MAX_FILENAME+1];
    SST* sst;
    MemTable* memtable;
    pthread_mutex_t my_mutex; //Dhlwnw ena mutex tis domis
    int writers; //Osoi theloun na grapsoun stin basi.
    int writing; //Osoi grafoun auti ti stigmi sti basi
    int reading; //Osoi diabazoun auti ti stigmi sti basi
    pthread_cond_t turn; //Dilwsi metablitis sinithikis
} DB;
```

```
DB* db_open_ex(const char* basedir, uint64_t cache_size)
{
    DB* self = calloc(1, sizeof(DB));

    pthread_mutex_init(&self->my_mutex,NULL); //Arxikopoisi tou mutex
    pthread_cond_init(&self->turn, NULL); //Arxikopoisi tis metablitis sinithikis.
    if (!self)
        PANIC("NULL allocation");

    strncpy(self->basedir, basedir, MAX_FILENAME);
    self->sst = sst_new(basedir, cache_size);

    Log* log = log_new(self->sst->basedir);
    self->memtable = memtable_new(log);

    return self;
}
```

Επιπλέον η μεταβλητή της συνθήκης αρχικοποήθηκε και στο αρχείο db.c εκεί όπου αρχικοποήθηκε και το Mutex που θα χρειαστούμε.

Ουσιαστικά η μεταβλητή συνθήκης δεν προστατεύει κρίσιμη περιοχή ή ένα κοινόχρηστο αλλά προστατεύει μια λειτουργία οπότε μπορούμε να πούμε ότι αποτελεί από μόνη της μια κρίσιμη περιοχή έτσι προσθέτουμε το mutex μας ώστε να την προστατεύσει.

```
int db_add(DB* self, Variant* key, Variant* value)
{
    int add_x; //metavliti pou xreiazomaste gia tin sosti leitourgia ths db_add
    pthread_mutex_lock(&self->my_mutex); //Kleidwma my_mutex prin ton elegxo metablitis tis sinithikis

    self->writers = self->writers + 1; //Auxanw kata 1 autous pou thelou na grapsoun

    while(self->reading > 0 || self->writing > 0 ){      //Oso kapoiow grafei H diabazei
        pthread_cond_wait(&self->turn, &self->my_mutex); //Tote perimene
    }

    self->writing = self->writing + 1; //Efoson mporw na synexisw ayxanw autous poy grafun kata 1

    pthread_mutex_unlock(&self->my_mutex); //to xekleidwnw wste na kanw to add

    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);
    }

    //An to kleidi moy eggrafei swsta pairnei tin timh 1 alliws tn timi 0.
    add_x = memtable_add(self->memtable, key, value);

    pthread_mutex_lock(&self->my_mutex); //Afou oloklirwthei h leitourgia kleidwnw to mutex

    self->writing = self->writing - 1; //Afairw oti auxisa parapanw
    self->writers = self->writers - 1; //Afairw oti auxisa parapanw afou i diadikasia oloklirwthike

    pthread_cond_broadcast(&self->turn); //Shmatodvtv ta upoloipa nimata oti h leitoyrgia oloklirwthike

    pthread_mutex_unlock(&self->my_mutex); //xekleidwma my_mutex

    return add_x;
}
```

Αρχικά, όπως αναφέρθηκε και παραπάνω κλειδώνουμε το mutex ώστε να προστατεύσουμε την μεταβλητή συνθήκης και προσαυξάνουμε την μεταβλητή writers , η οποία αναφέρεται σε όσες λειτουργίες επιθυμούν να γράψουν στην βάση.

Στη συνέχεια ακολουθεί ο έλεγχος αν υπάρχει κάποιος που γράφει ή διαβάζει ήδη στην βάση οπότε θα πρέπει να καλέσουμε την wait ώστε το νήμα που θέλει να γράψει να περιμένει σε μια μεταβλητή συνθήκης έως ότου η μεταβλητή το ενημερώσει ότι μπορεί να συνεχίσει.

Από την στιγμή που το νήμα περιμένει δεν χρειάζεται συνεχής επανέλεγχος, μέχρι να ειδοποιηθεί το νήμα από την κλήση broadcast ώστε να ξυπνήσει το νήμα. Έτσι εφόσον δεν βρεθεί κάποιος να γράφει στη βάση ή να διαβάζει αυξάνουμε τον αριθμό αυτών που γράφουν στη βάση κατά ένα και ξεκλειδώνουμε το mutex ο ώστε να γίνει η λειτουργία εγγραφής.

Έτσι αφού ολοκληρωθεί η εγγραφή ξανά κλειδώνουμε το mutex έτσι ώστε να ανανεώσουμε τις μεταβλητές και να ειδοποιήσουμε τη μεταβλητή συνθήκης έτσι ώστε τα υπόλοιπα νήματα να μπορέσουν να συνεχίσουν .

```
int db_get(DB* self, Variant* key, Variant* value)
{
    int get_x; //metavliti pou xreiazomaste gia tin sosti leitourgia ths db_get

    pthread_mutex_lock(&self->my_mutex); //Kleidwma my_mutex prin ton elegxo metablitis tis sinithikis

    while(self->writers > 0){   //An uparxei kapoios pou thelei na grapsei perimenw
                                //Protairaiotita stous grafeis

        pthread_cond_wait(&self->turn, &self->my_mutex);//Tote perimene
    }

    self->reading = self->reading + 1; //Afou mporw na diabasw ayxanw tin metabliti kata 1

    pthread_mutex_unlock(&self->my_mutex); //to xekleidwnw wste na kanw to get

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

    pthread_mutex_lock(&self->my_mutex); //Afou oloklirwthei h leitourgia kleidwnw to mutex
    self->reading = self->reading - 1;   //Afairw oti auxisa parapanw

    pthread_cond_broadcast(&self->turn); //Shmatodvtv ta upoloipa nimata oti h leitoyrgia oloklirwthike

    pthread_mutex_unlock(&self->my_mutex); //xekleidwma my_mutex

    return get_x;
}
```

Η ίδια η υλοποίηση έχει γίνει και στην συνάρτηση db_get με την **μονή σημαντική διαφορά ότι ελέγχει αν κάποιος περιμένει να γράψει στη βάση και δεν ελέγχουμε αν κάποιος διαβάζει ,** αν υπάρχει κάποιος εγγραφέας που περιμένει τότε η ίδια δεν θα εκτελεστεί και θα περιμένει και θα **δώσει προτεραιότητα στον γραφέα**. Εάν δεν υπάρχει γραφέας που περιμένει τότε θα εκτελεστεί η λειτουργία ανάγνωσης με παρόμοιο τρόπο όπως και παραπάνω.

Με τη μεταβλητή συνθήκης εξασφαλίζουμε ότι μόνο ένας θα γράφεις ή θα διαβάζει από τη βάση αφού ή άλλη λειτουργία θα ενημερώνεται από αυτή και όλοι οι υπόλοιποι θα περιμένουν εξασφαλίζοντας έτσι τον αμοιβαίο αποκλεισμό.

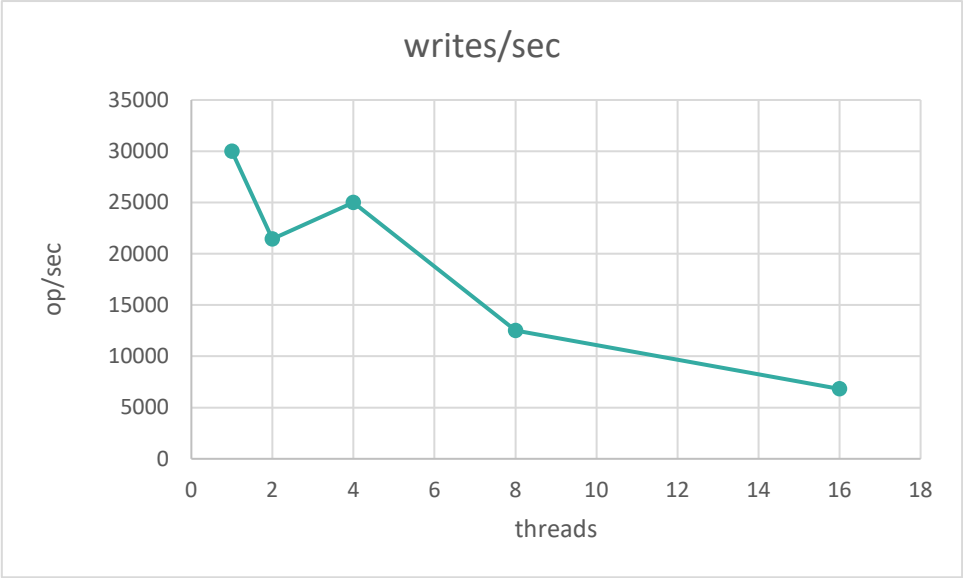
Πηγή:

https://csresources.github.io/SystemProgrammingWiki/SystemProgramming/Synchronization,-Part-7:-The-Reader-Writer-Problem/

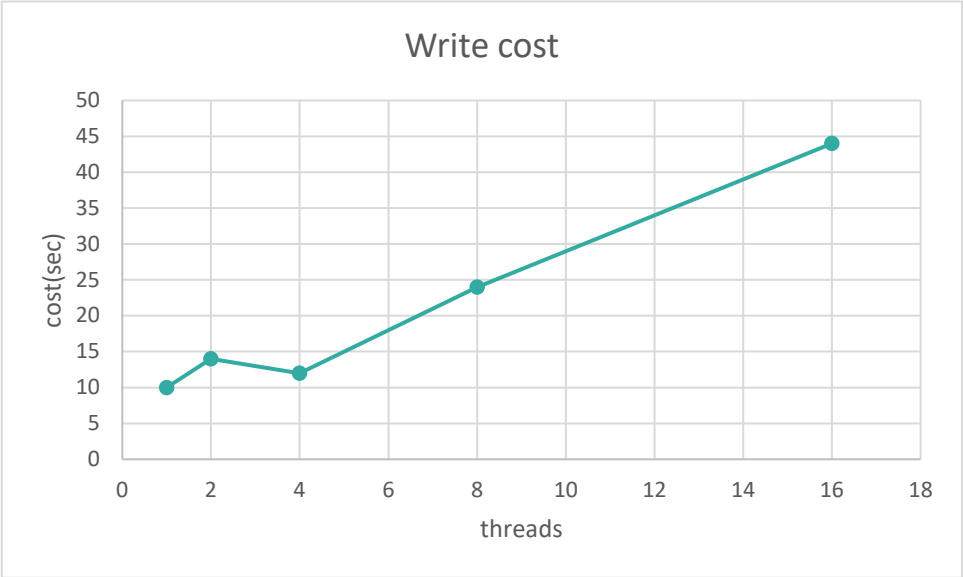
Αποτελέσματα υλοποίησης αναγνωστών - γραφέων.

Αρχικά θα ξεκινήσουμε με τη **λειτουργία εγγραφής** διότι όπως είδαμε παραπάνω με την ύπαρξη πολλαπλών νημάτων χωρίς κάποιο είδος ταυτοχρονισμού η λειτουργία δεν τερμάτιζε ποτέ όπως ήταν αναμενόμενο. Τώρα με την νέα υλοποίηση η λειτουργία εγγραφής δουλεύει σωστά και έτσι έχουμε τρέξει **πειράματα για 300.000 εισόδους** και για πολλαπλά νήματα.

threads	writes/sec
1	30000
2	21428
4	25000
8	12500
16	6818



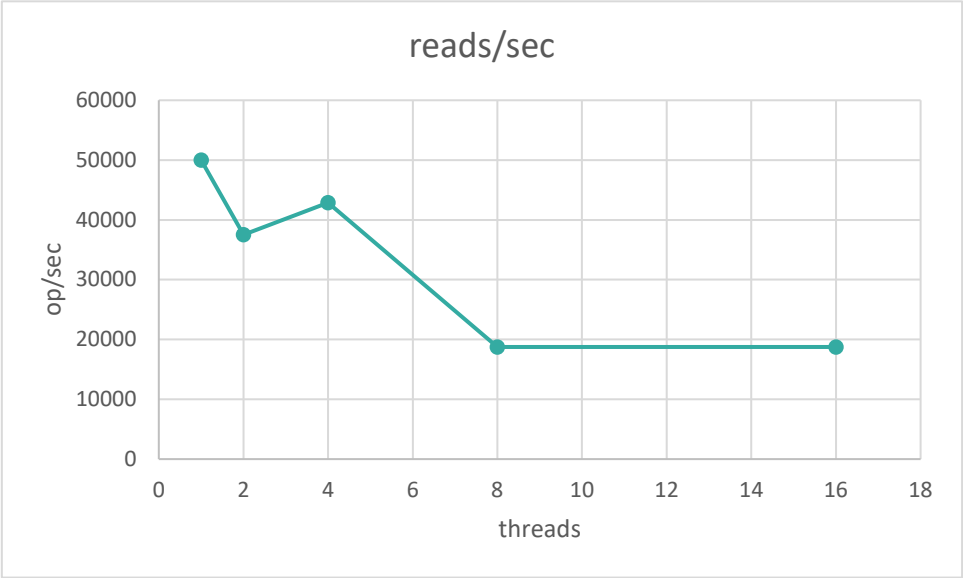
threads	cost
1	10
2	14
4	12
8	24
16	44



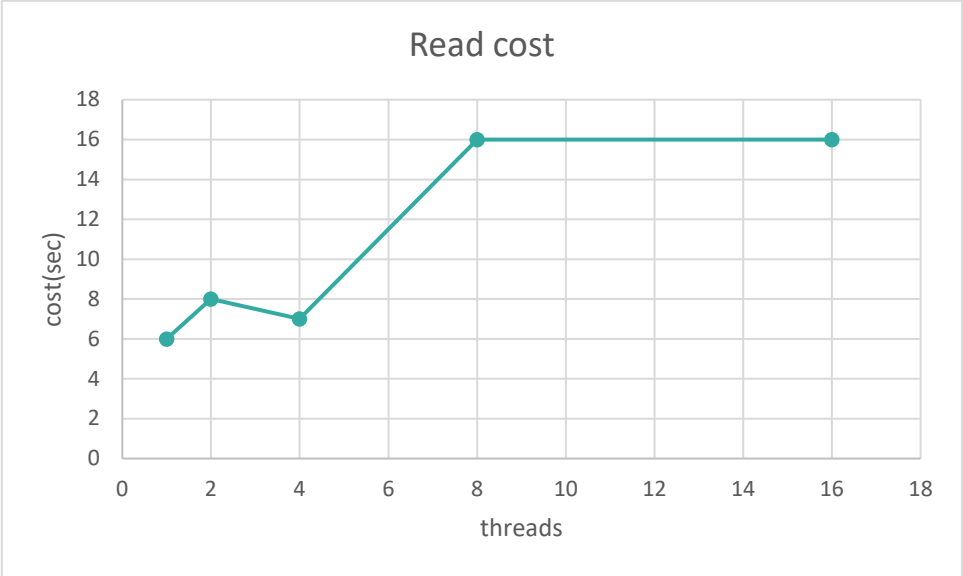
Τα αποτελέσματα είναι τα αναμενόμενα έχουμε μια αύξηση στο κόστος όσο αυξάνουμε τα νήματα αλλά σε σχέση με την προηγούμενη υλοποίηση μας έχουμε καλύτερο χρόνο θα αναφερθούμε εκτενώς παρακάτω.

Στη συνέχεια εκτελέσαμε πειράματα για την λειτουργία ανάγνωσης όπως κάναμε και στην αρχή όπου υλοποιήσαμε την πολυνηματική λειτουργία. Ξανά, το πλήθος των αναγνώσεων είναι 300.000 και κάναμε πειράματα για πολλαπλά νήματα με την νέα μας υλοποίησή.

threads	reads/sec
1	50000
2	37500
4	42857
8	18750
16	18750



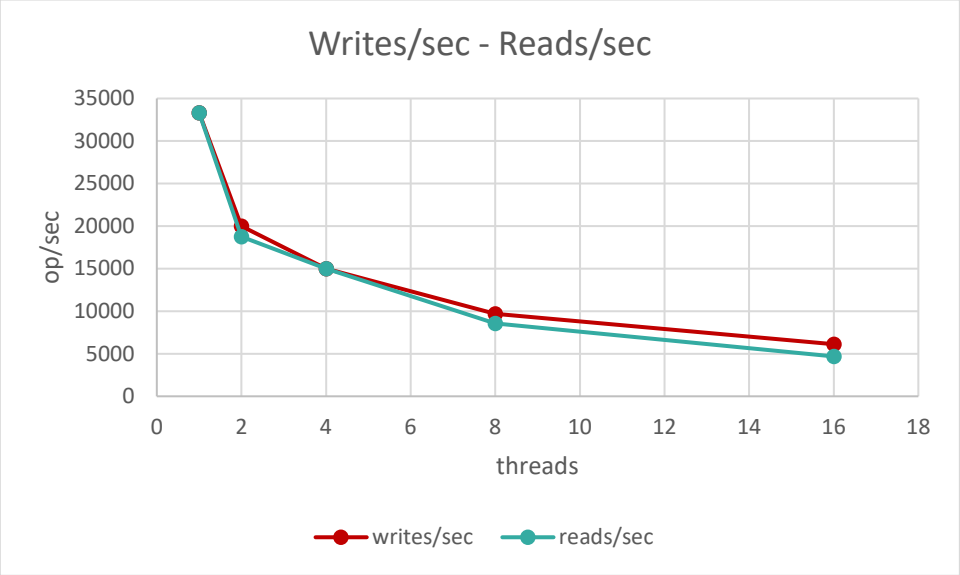
threads	cost
1	6
2	8
4	7
8	16
16	16



Και στη λειτουργία των αναγνώσεων παρατηρείται βελτίωση στους χρόνους όσον αφορά την τους χρόνους με την καθολική κλειδαριά.

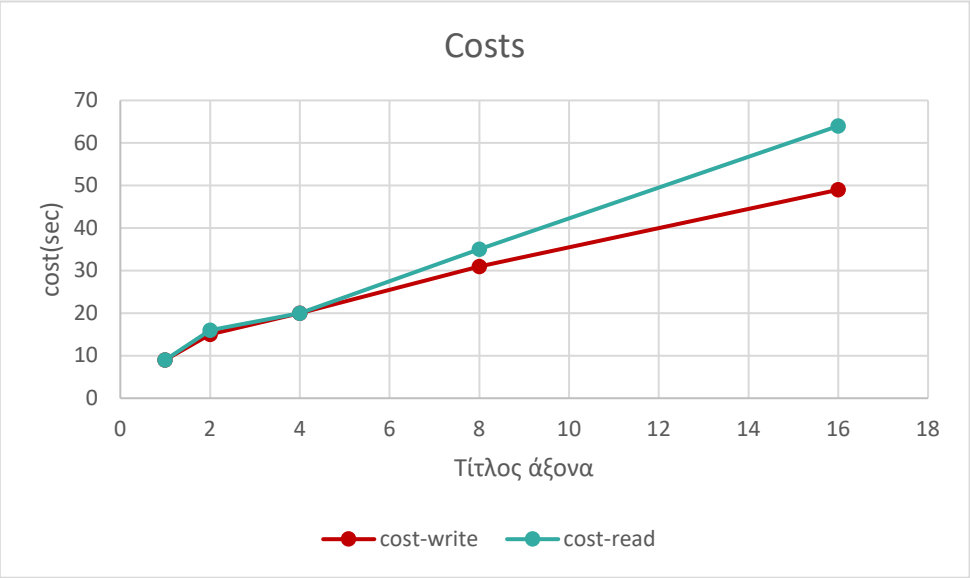
Επιπλέον κάναμε πειράματα για τη λειτουργία writeread με 600.000 εισόδους και 50% ποσοστό ώστε να κάνει 300.000 εγγραφές και 300.000 αναγνώσεις.

threads	writes/sec	reads/sec
1	33333	33333
2	20000	18750
4	15000	15000
8	9677	8571
16	6122	4687



Παρατηρούμε ότι γίνονται περισσότερες εγγραφές ανά δευτερόλεπτο αφού οι γραφείς έχουν προτεραιότητα στην υλοποίηση μας.

threads	cost-write	cost-read
1	9	9
2	15	16
4	20	20
8	31	35
16	49	64



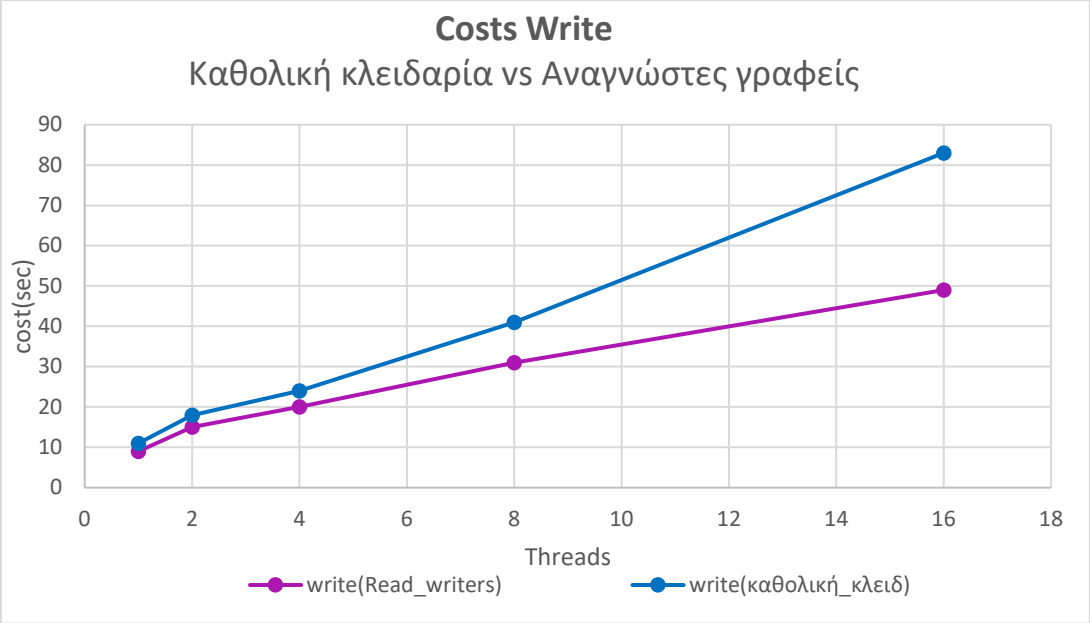
Εδώ μπορούμε εύκολα να παρατηρήσουμε ότι η λειτουργία της ανάγνωσης έχει υψηλότερο κόστος από τη λειτουργία της εγγραφής αυτό συμβαίνει διότι η υλοποίηση μας δίνει προτεραιότητα στους γραφείς και έτσι έστω και ένας γραφέας να θέλει να γράψει τη βάση οι αναγνώστες πρέπει να περιμένουν με συνέπεια το αυξημένο κόστος των λειτουργιών ανάγνωσης.

Καθολική κλειδαριά vs Readers – writers.

Κρίναμε εύστοχο να συγκρίνουμε τα αποτελέσματα της καθολικής κλειδαριάς με την υλοποίηση που κάναμε για το πρόβλημα αναγνωστών γραφέων έτσι ώστε να συγκρίνουμε το επίπεδο ταυτοχρονισμού που μας παρέχουν . Όπως ήταν αναμενόμενο η δεύτερη υλοποίηση μας δηλαδή η υλοποίηση των αναγνωστών γραφέων δίνει αρκετά καλύτερα επίπεδα ταυτοχρονισμού από ότι η ύπαρξη καθολικής κλειδαριάς. Έτσι εκτελέσαμε πειράματα με 600.000 εισόδους για τη λειτουργία writeread με ποσοστό 50% και πολλαπλά νήματα.

Αρχικά για την λειτουργία εγγραφής έχουμε μικρότερο κόστος από ότι αυτό που είχαμε με την καθολική κλειδαριά και έχουμε περισσότερες εγγραφές ανά δευτερόλεπτο.

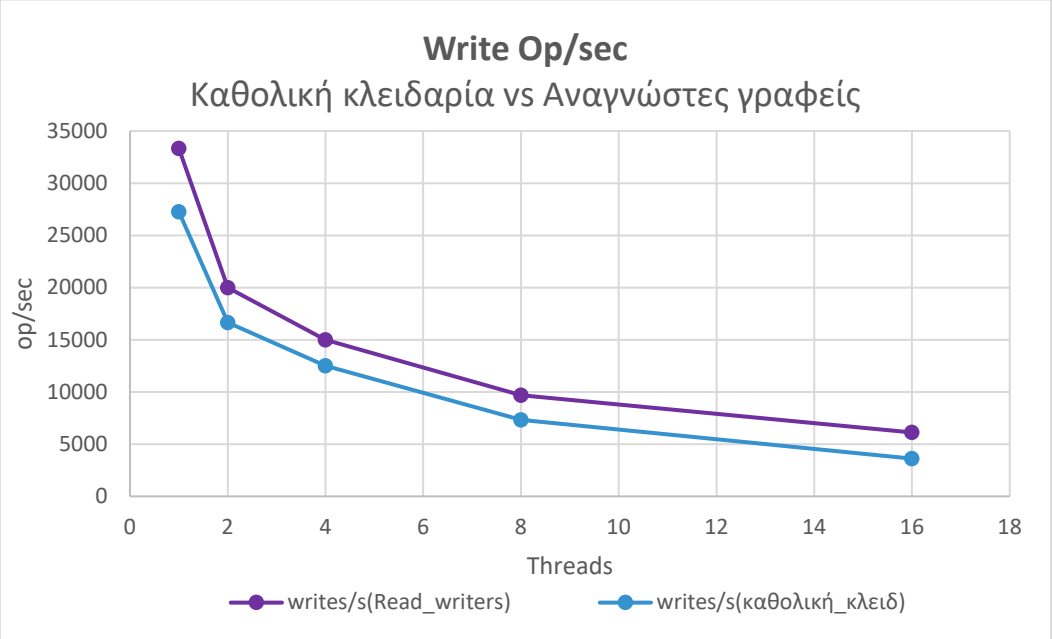
threads	write(Read_writers)	write(καθολική_κλειδ)
1	9	11
2	15	18
4	20	24
8	31	41
16	49	83



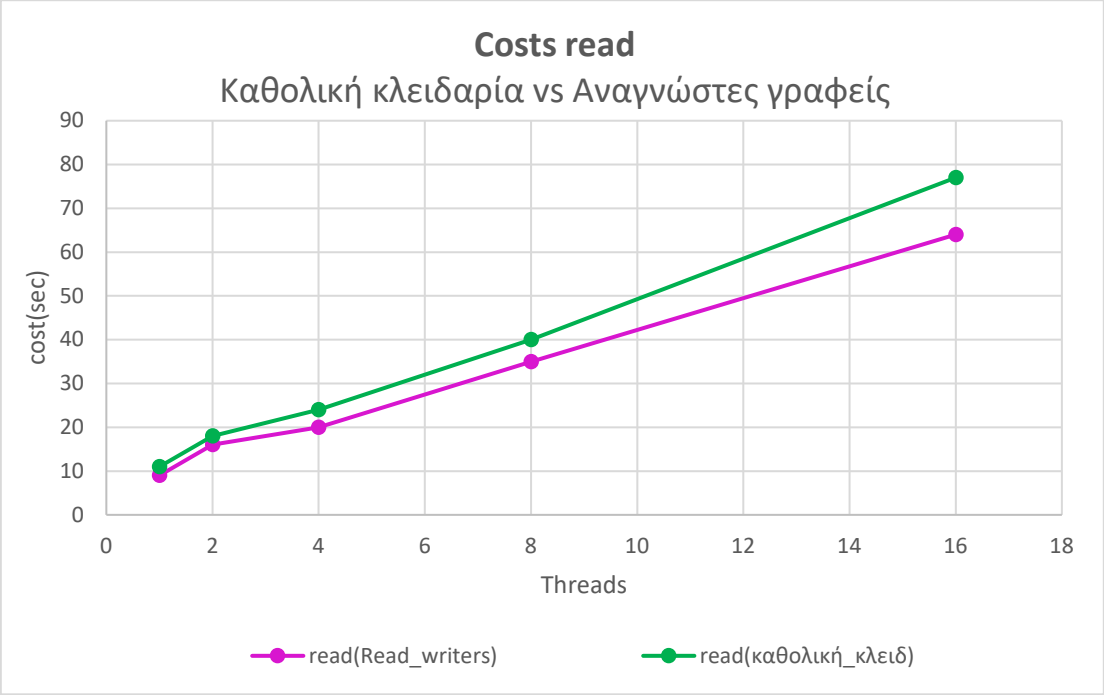
Όπως φαίνεται παραπάνω έχουμε μικρότερο κόστος αφού έχουμε επιτύχει καλύτερο επίπεδο ταυτοχρονισμού και δίνουμε προτεραιότητα στους γραφείς μας

Επίσης πετυχαίνουμε περισσότερες λειτουργίες ανά δευτερόλεπτο αφού δεν έχουμε τις καθυστερήσεις λόγω ύπαρξη της καθολικής κλειδαριάς που θα μου κλείδωνε τη λειτουργία εγγραφής εάν ήθελε έστω και ένας αναγνώστης να διαβάσει τη βάση αφού έχω προτεραιότητα στους εγγραφείς.

threads	writes/s(Read_writers)	writes/s(καθολική_κλειδ)
1	33333	27272
2	20000	16666
4	15000	12500
8	9677	7317
16	6122	3614

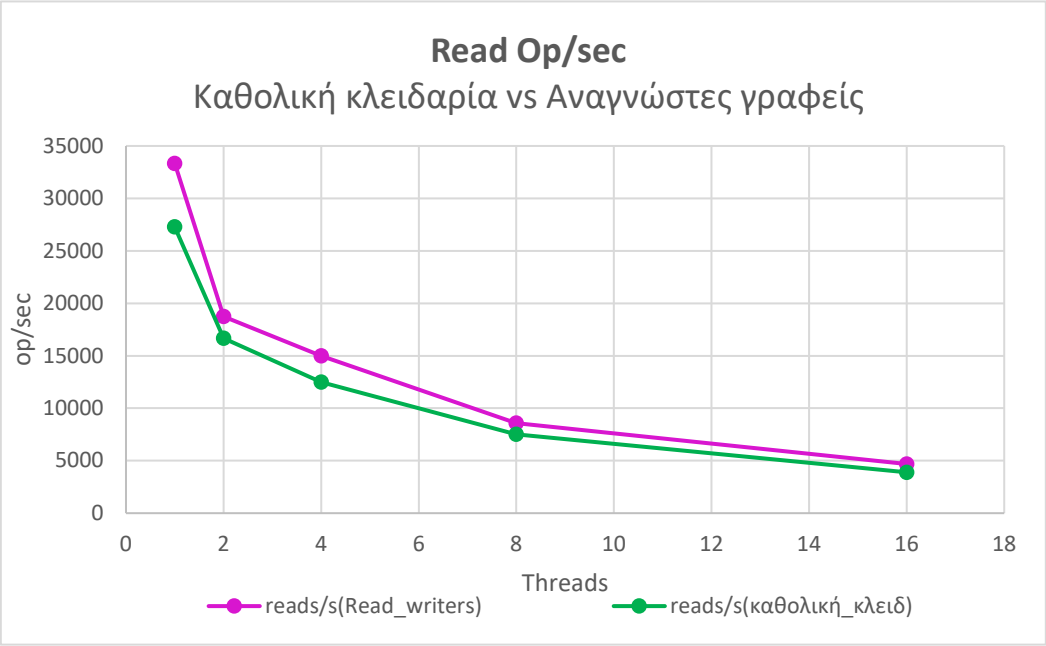


threads	read(Read_writers)	read(καθολική_κλειδ)
1	9	11
2	16	18
4	20	24
8	35	40
16	64	77



Και εδώ παρατηρούμε μείωση του κόστους στις εγγραφές αυτό συμβαίνει διότι πετυχαίνουμε καλύτερο επίπεδο ταυτοχρονισμού.

threads	reads/s(Read_writers)	reads/s(καθολική_κλειδ)
1	33333	27272
2	18750	16666
4	15000	12500
8	8571	7500
16	4687	3896



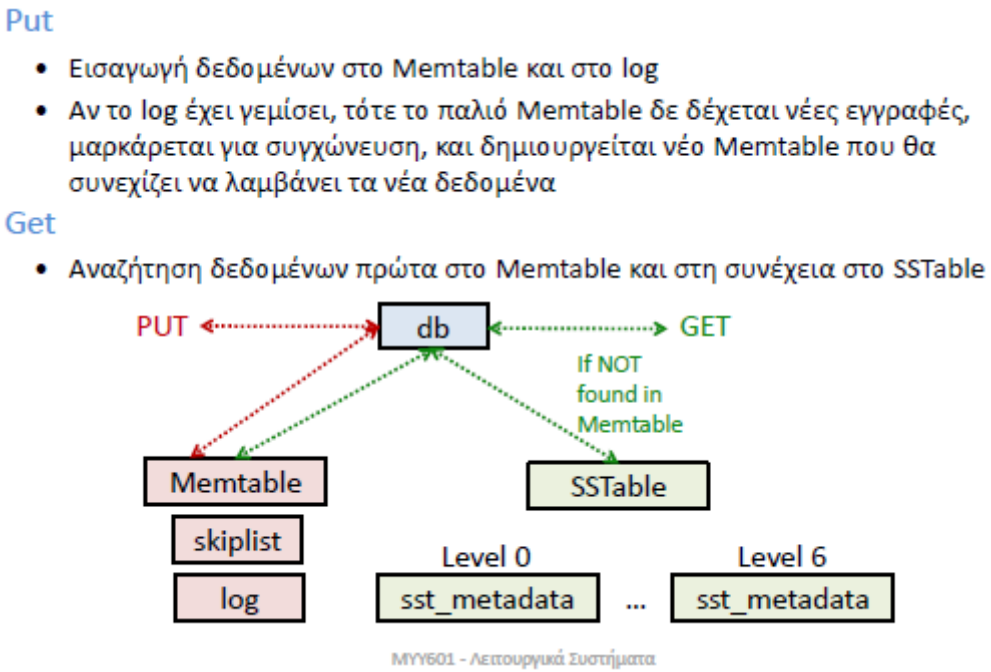
Γίνονται περισσότερες αναγνώσεις ανά δευτερόλεπτο λόγω του καλύτερου ταυτοχρονισμού που πετύχαμε.

Έξοδο της τελικής εντολής make.

Όπως αναφέρεται στην εκφώνηση της άσκησης παρακάτω βλέπουμε την έξοδο από την εντολή make all , όλα τα αρχεία μας μεταφράζονται επιτυχώς.

```
myy601@myy601lab1:~/kiwi/kiwi-source$ make all
cd engine && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/engine'
    CC db.o
    CC memtable.o
    CC indexer.o
    CC sst.o
    CC sst_builder.o
    CC sst_loader.o
    CC sst_block_builder.o
    CC hash.o
    CC bloom_builder.o
    CC merger.o
    CC compaction.o
    CC skiplist.o
    CC buffer.o
    CC arena.o
    CC utils.o
    CC crc32.o
    CC file.o
    CC heap.o
    CC vector.o
    CC log.o
    CC lru.o
    AR libindexer.a
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/engine'
cd bench && make all
make[1]: Entering directory '/home/myy601/kiwi/kiwi-source/bench'
gcc -g -ggdb -Wall -Wno-implicit-function-declaration -Wno-unused-but-set-variab
le bench.c kiwi.c -L ../engine -lindexer -lpthread -lsnappy -o kiwi-bench
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
```

Η σκέψη μας για μια βελτίωση της υλοποίησης readers/writers.



Σύμφωνα με την παραπάνω λειτουργία των λειτουργιών put και get σκεφτήκαμε ότι μια καλύτερη υλοποίηση θα επέτρεπε συνεχώς να γίνονται εγγραφές στο memtable,το ίδιο να συμβαίνει και για τη λειτουργία get όταν αυτή θέλει να διαβάσει κάποιο κλειδί που περιέχεται σε κάποιο αρχείο SST.θα πρέπει να ελέγχουμε για ποιο κλειδί ψάχνει η λειτουργία ανάγνωσης ώστε να μην κλειδώνουμε το SSTable ,αν τώρα η λειτουργία θέλει να διαβάσει αυτό από το memtable θα πρέπει να περιμένει να τελειώσουν όλες οι εγγραφές αφού θα δώσουμε και πάλι προτεραιότητα στους γραφείς εάν δεν γίνεται κάποια εγγραφή ή δεν περιμένει κάποιος γραφέας να γράψει σε αυτό τότε δεν θα κλειδώνουμε το memtable και θα μπορεί να διαβάσει.

Όμως πρέπει να ελέγχουμε και την περίπτωση που γίνεται κάποιο merge αφού υπάρχει ένα νήμα που τρέχει στο background που κλειδώνει και εκτελεί αυτές τις εργασίες μέσα στα αρχεία sst.c , memtable.c. Αυτό το νήμα δεν μπορεί να τρέχει παράλληλα με κάποια λειτουργία είτε αυτή είναι λειτουργία put είτε είναι get.