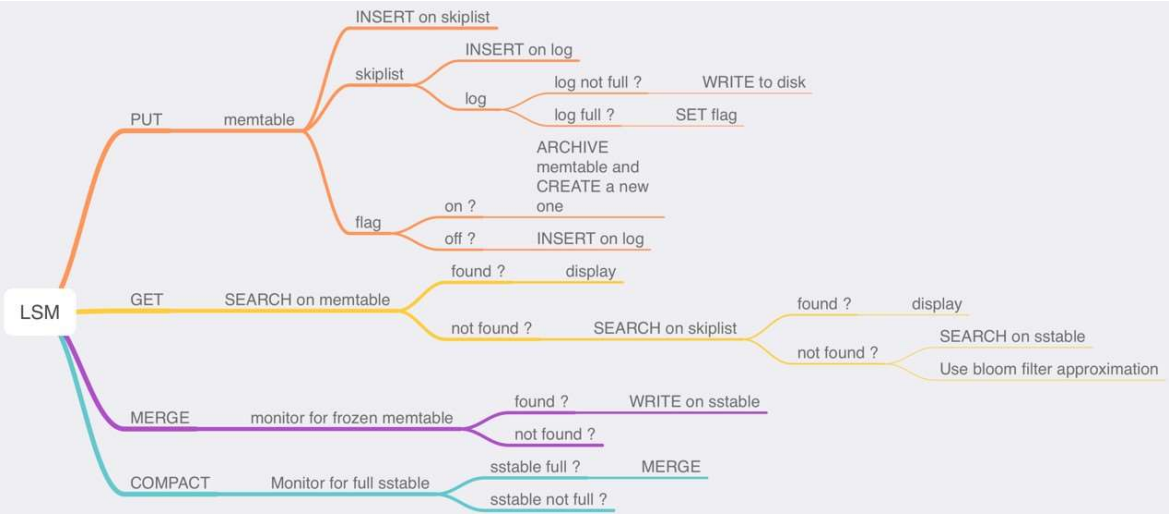
Λειτουργικά Συστήματα, 1η εργαστηριακή άσκηση, 2022

Αθανάσιος Παπαποστόλου, 4147
Αθανάσιος Κουρέας, 4392

Στη άσκηση θα υλοποιήσουμε πολυνηματική λειτουργία στην μηχανή αποθήκευσης Kiwi, που βασίζεται σε δέντρο LSM. Συγκεκριμένα θα εξασφαλίσουμε ορθότητα στην λειτουργία πολλών νημάτων στις εντολές put και get που παρέχει η μηχανή. Θα διατηρήσουμε στατιστικά χρόνου εκτέλεσης και θα παρουσιάσουμε τις βελτιώσεις σε σε χρόνο εκτέλεσης.

1. Περιγραφή Αλλαγών

Αρχικά προσπαθούμε να κατανοήσουμε το πλαίσιο των λειτουργιών και να βρούμε ποια σημεία θα προκαλέσουν πιθανά προβλήματα σε ασύγχρονη λειτουργία.



Όπως φαίνεται από το mindmap, οι βασικές λειτουργίες είναι οι PUT, GET, MERGE, COMPACT, και οι βασικοί πόροι το γενικό `memtable`, που χρησιμοποιεί ένα `skiplist` self balancing, το temporary `log`, και τα `sstables` για την εισαγωγή στον δίσκο. Επίσης, στο αρχικό πρόγραμμα φαίνεται πως η διαδικασία του MERGE, που χρησιμοποιεί το sstable είναι ήδη thread safe. Επομένως πετυχαίνουμε μέγιστο ταυτοχρονισμό με correctness στις δομές δεδομένων, αναβαθμίζοντας τις υπόλοιπες λειτουργίες, και συγκεκριμένα σύμφωνα με το διάγραμμα αυτές γραμμένες σε κεφαλαία, ως thread safe.

Αναλύοντας περισσότερο παρατηρούμε πως σαν κρίσιμους πόρους έχουμε τα skiplist, log, flag, και sstable. Ιδανικά, θα θέλαμε να προστατεύσουμε την επεξεργασία αυτών των δομών για συνεπές αποτέλεσμα, στην εισαγωγή και εξαγωγή. Με ξεφύλλισμα του κώδικα βλέπουμε πως το macro BACKGROUND_MERGE προστατεύει σε γενικές γραμμές το sstable, ωστόσο θα φανεί αργότερα εάν μπορεί να βελτιωθεί παραπάνω. Στη συνέχεια ακολουθώντας το callback path, στο API του db, βρίσκουμε τις εξής βασικές κρίσιμες περιοχές.

1) db_open → db_open_ex → sst_new → _read_manifest → _schedule_compation
2) db_open → db_open_ex → memtable_new → skiplist_new → (refcount)
3) db_open → db_open_ex → memtable_new → skiplist_acquire → (refcount, hdr)
4) db_open → db_open_ex → memtable_new → log_recovery → _load_from → skiplist_insert → (Several fields of the SkipList pointer)
5) db_close → sst_merge → (Reading fields)
6) db_close → skiplist_release → (refcount, hdr, forward)
7) db_close → sst_free → _write_manifest
8) db_close → memtable_free → skiplist_release (refcount, hdr) → (arena_free Pool*)
9) db_add → memtable_needs_compaction → (Reading fields)
10) db_add → sst_merge → (Reading fields)
11) db_add → memtable_reset → skiplist_release / skiplist_acquire

12) db_add → memtable_add → _memtable_edit → (add_count)
13) db_add → memtable_add → _memtable_edit → log_append (all calls after that)
14) db_add → memtable_add → _memtable_edit → skiplist_insert (fields, arena_alloc)
15) db_get → memtable_get → skiplist_lookup → (fields)

16) db_get → sst_get → (calls and fields)

Με αυτή την ανάλυση αποδεικνύουμε πως κάποια σημεία είναι ήδη προστατευμένα με mutual exclusion, συγκεκριμένα τα 1-11, και 16. Επομένως προσθέτουμε τα κατάλληλα mutexes στα σημεία 12-15, εκεί δηλαδή που πραγματοποιούνται οι ουσιαστικές εισαγωγές και εξαγωγές. Δεν σημειώνουμε κλήσεις των printf, snprintf, και τις διάφορες κλήσεις για memory allocation (malloc, free), καθώς αυτές με το -pthread flag γίνονται thread safe. Πιθανόν να υπάρχουν και πιο συγκεκριμένες κρίσιμες περιοχές που δεν βρήκαμε.

Χρειάζεται να προσέξουμε τα mutexes να μην πραγματοποιούν lock και unlock μέσα σε επαναλήψεις για να μην δημιουργήσουμε μεγάλο processing hog. Προσέχουμε επίσης να μην δημιουργούμε locks σε σημεία που εκτελούν πολύ μικρές λειτουργίες (όπως to vector ή το arena), καθώς θα ήταν προτιμότερο να έχουμε mutual exclusion σε πιο αφαιρετικές συναρτήσεις που περιλαμβάνουν τις ειδικές λειτουργίες με σκοπό την βελτίωση της ταχύτητας, και για να σιγουρέψουμε την συνέπεια των δομών.

Έχοντας κατανοήσει το εύρος των αλλαγών του state, υποθέτοντας ότι το πρόγραμμα είναι ολοκληρωμένα thread safe, σχεδιάζουμε το πλάνο των test ως εξής. Συγκρίνουμε τους χρόνους για:

a) Πολλαπλά read (σειριακά και ασύγχρονα).
b) Πολλαπλά write (σειριακά και ασύγχρονα).
c) X read και X write (σειριακά) με X read/write (ασύγχρονα και ταυτόχρονα).
d) X % read και (100-X)% write (σειριακά) και αντίστοιχα read/write.

Το ποσοστιαίο distribution των threads το ορίσουμε σε δεκάδες, πχ με ένα ποσοστό 40% read να ξεκινήσουν 4 threads για read και 6 threads για write.

2. Υλοποίηση

Αρχικά πραγματοποιούμε κάποιες quality of life μετατροπές στον κώδικα του `bench.c` κάνοντας refactor κομμάτια που περιπλέκουν συνολικά την κατανόηση. Εκεί δεν προσθέτουμε καινούργιο κώδικα αλλά κυρίως εξαλείφουμε τα διπλότυπα έτσι ώστε να έχουμε ένα ομοιόμορφο template για τα καινούρια test. Ωστόσο βάζουμε ένα ακόμη header file που ορίζει τις test συναρτήσεις.

```
#ifndef __KIWI_H_
#define __KIWI_H_

void _write_test(long int count, int r);
void _read_test(long int count, int r);

#endif
```

Ξεκινάμε να μετατρέψουμε σε thread-safe το σημείο 12 στο memtable.c : 52 στην συνάρτηση _memtable_edit. Παρατηρούμε ότι όλες οι γραμμές που αλλάζουν κάποιο πεδίο του self (MemTable*) είναι κρίσιμες περιοχές (από την γραμμη 82). Η πρόσβαση στο `needs_compaction` θα προστατευτεί μεμονομένα στο log_append() και όμοια με την skiplist_insert επομένως τοποθετούμε mutex γύρω από το block κώδικα στο οποίο χρησιμοποιείται το add_count. Χρησιμοποιώντας την τεχνική του `sst.c` προσθέτουμε το lock ως πεδίο στο MemTable struct. Επίσης υλοποιούμε το convention του BACKGROUND_MERGE macro, έτσι προσθέτουμε το προαιρετικό BACKGROUND_PUT macro και το χρησιμοποιούμε εκεί που πραγματοποιούνται τα κλειδώματα. Στην constructor συνάρτηση κάνουμε δυναμικά το initialize των mutex.

memtable.h : 17

```
    uint32_t needs_compaction;
    uint32_t del_count;
    uint32_t add_count;
#ifdef BACKGROUND_PUT
    pthread_mutex_t edit_lock;
#endif
} MemTable;
```

config.h : 54

```
#define BACKGROUND_MERGE
#define WITH_SNAPPY

#define BACKGROUND_PUT
#define BACKGROUND_GET

#endif
```

memtable.c : 12

```
MemTable* memtable_new(Log* log)
{
#ifndef BACKGROUND_PUT
    pthread_mutex_init(&self->edit_lock, NULL);
#endif

    MemTable* self = malloc(sizeof(MemTable));
```

: 92

```
#ifdef BACKGROUND_PUT
    pthread_mutex_lock(&self->edit_lock);
#endif

    if (opt == ADD)
        self->add_count++;
    else
        self->del_count++;

#ifdef BACKGROUND_PUT
    pthread_mutex_unlock(&self->edit_lock);
#endif
```

Στη συνέχεια μετατρέπουμε το σημείο 13 σε thread safe. Βλέπουμε ότι πρέπει να προστατέψουμε το file_length (log.c : 162), καθώς και την διαδικασία του file_append_raw. Προσθέτουμε το mutex στο struct definition του Log και όπως και παραπάνω χρησιμοποιούμε το BACKGROUND_PUT macro για το initialization στο log_new και το lock και unlock στην log_append. Στην log_append βάζουμε το αποτέλεσμα που επιστρέφει η συνάρτηση με το return σε μία local μεταβλητή έτσι ώστε να το συμπεριλάβουμε στο lock/unlock block.

log.h : 13

```
    char basedir[MAX_FILENAME];
#ifdef BACKGROUND_PUT
    pthread_mutex_t append_lock;
#endif
} Log;
```

log.c : 21

```
Log* log_new(const char *basedir)
{
    Log* self = calloc(1, sizeof(Log));

#ifdef BACKGROUND_PUT
    pthread_mutex_init(&self->append_lock, NULL);
#endif
```

: 181

```
    int result;

#ifdef BACKGROUND_PUT
    pthread_mutex_lock(&self->append_lock);
#endif

    // Here we should instruct the File class to do some fsync
after
    // a certain amount of insertions.
    file_append_raw(self->file, value, length);

    self->file_length += length;
    result = (self->file_length >= LOG_MAXSIZE);

#ifdef BACKGROUND_PUT
    pthread_mutex_unlock(&self->append_lock);
#endif

    return result;
}
```

Μετά μετατρέπουμε το σημείο 14 σε thread safe. Προσθέτουμε στο struct του skiplist.h το mutex ως πεδίο, κάνουμε initialize στο heap το mutex στην skiplist_new, και μετά περικλείουμε στο lock/unlock block την skiplist_insert. Φροντίζουμε να κάνουμε unlock σε κάθε περίπτωση στα return statements, έτσι ώστε να μην προκύψει spinlock. Χρησιμοποιούμε και πάλι την τεχνική των guard macros γύρω από τα mutexes.

skiplist.h : 37

```
#ifdef BACKGROUND_MERGE
    pthread_mutex_t lock;
    int refcount;
#endif

#ifdef BACKGROUND_PUT
    pthread_mutex_t insert_lock;
#endif
```

skiplist.c : 19

```
SkipList* skiplist_new(size_t max_count)
{
    int i;
    SkipList* self = calloc(1, sizeof(SkipList));

#ifdef BACKGROUND_PUT
    pthread_mutex_init(&self->insert_lock, NULL);
#endif
```

: 109

```
int skiplist_insert(SkipList* self, const char *key, size_t
klen, OPT opt, char *data)
{
#ifdef BACKGROUND_PUT
    pthread_mutex_lock(&self->insert_lock);
#endif
    int i, new_level;
    SkipNode* update[SKIPLIST_MAXLEVEL];
    SkipNode* x;
```

: 136

```
        free(tmp);

        #ifdef BACKGROUND_PUT
            pthread_mutex_unlock(&self->insert_lock);
        #endif
        return STATUS_OK;
    }
```

: 168

```
    for (i = 0; i <= new_level; i++)
    {
        x->forward[i] = update[i]->forward[i];
        update[i]->forward[i] = x;
    }

#ifdef BACKGROUND_PUT
    pthread_mutex_unlock(&self->insert_lock);
#endif
    return STATUS_OK;
```

Μετατρέπουμε το σημείο 15 σε thread safe. Βλέπουμε ότι η κρίσιμη περιοχή στο memtable_get (memtable.c : 117) περιορίζεται γύρω από την κλήση της skiplist_lookup. Στο skiplist.c : 194 στην συνάρτηση η πρόσβαση στα πεδία hdr, level, forward καθώς και η χρήση του SkipNode* πρέπει να προστατευτεί. Προσέχουμε να κάνουμε unlock πριν από κάθε return να αποφύγουμε τα spinlocks.

skiplist.h : 37

```
#ifdef BACKGROUND_GET
    pthread_mutex_t get_lock;
#endif

    // the data structure
    SkipNode* hdr;
    Arena* arena;
} SkipList;
```

skiplist.c : 197

```
#ifdef BACKGROUND_GET
    pthread_mutex_init(&self->get_lock, NULL);
#endif

    if (!self)
        PANIC("NULL allocation");
```

```
    int i;
#ifdef BACKGROUND_GET
    pthread_mutex_lock(&self->get_lock);
#endif
```

```
    SkipNode* x = self->hdr;

    for (i = self->level; i >= 0; i--)
    {
        while (x->forward[i] != self->hdr &&
               cmp_lt(x->forward[i]->data, key, klen))
            x = x->forward[i];
    }

    x = x->forward[0];
    if (x != self->hdr && cmp_eq(x->data, key, klen)) {
        #ifdef BACKGROUND_GET
            pthread_mutex_unlock(&self->get_lock);
        #endif
        return x;
    }

    #ifdef BACKGROUND_GET
        pthread_mutex_unlock(&self->get_lock);
    #endif
    return NULL;
```

Εφόσον έχει επιτευχθεί το thread safety συνολικά στο πρόγραμμα (αν και ίσως όχι αναγκαστικά στο βέλτιστο επίπεδο), μπορούμε να γράψουμε τα test για την ταυτόχρονη λειτουργία.

Για να γράψουμε τα τεστ, αλλάζουμε τον κώδικα του kiwi.c έτσι ώστε να απομονώσουμε τις διαδικασίες που χρειάζονται ταυτοχρονισμό. Δεν θα βάλουμε όλον τον κώδικα στην αναφορά καθώς τον έχουμε αλλάξει σχεδόν ολοκληρωτικά, αλλά θα τον περιγράψουμε αναλυτικά.

Αρχικά σπάμε τις συναρτήσεις _read_test και _write_test σε λογικά blocks όπου τοποθετούμε σε συναρτήσεις. Αναλυτικότερα, ομαδοποιούμε τις απαιτούμενες μεταβλητές που χρησιμοποιούν τα τεστ και τις βάζουμε σε συγκεκριμένα structs (read_data_t, και write_data_t). Στην αρχή του κάθε τεστ αρχικοποιούμε τις μεταβλητές αυτές στις (get_read_initializers, και get_write_initializers) αντίστοιχα. Προσέχουμε τα πεδία που ορίζονταν σαν πίνακες στο stack να τις μετατρέψουμε σε δείκτες στο heap.

```
// char key[KSIZE + 1];
    char *key = (char*)malloc(sizeof(char) * (KSIZE + 1));
```

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

```
    char *key = (char*)malloc(sizeof(char) * (KSIZE + 1));
    char *val = (char*)malloc(sizeof(char) * (VSIZE + 1));
    char *sbuf = (char*)malloc(1024);
```

Αφού έχουμε τον δείκτη με τα στοιχεία τον χρησιμοποιούμε για να εκτελέσουμε τις διαδικασίες σύμφωνα με τα τις απαιτήσεις που θέσαμε. Προσέχουμε να ανοίγουμε την βάση με την db_open μόνο μία φορά πριν την αρχικοποίηση των threads έτσι ώστε να αποφύγουμε τα πολλαπλά db_open και db_close, καθώς και την ανάγκη για mutex locking σε αυτές τις συναρτήσεις.

```
/* Open db and execute the writes */
    /* Makes sure opening and closing the db happens in 1 thread
*/
    initializers->db = db_open(DATAS);
        cost = execute_and_time_write(initializers);
    db_close(initializers->db);
```

Οι (execute_and_time_write, και execute_and_time_read) εκτελούν τις επαναλήψεις για προσθήκη και εξαγωγή στοιχείων. Οι συναρτήσεις έχουν υλοποιηθεί με ιδιαίτερο τρόπο, έτσι ώστε να καλύπτουν τόσο της περιπτώσεις της μονονηματικής εκτέλεσης όσο και της πολυνηματικής. Χρησιμοποιούμε το buffer_size για να υπολογίσουμε το εύρος του αριθμού των στοιχείων που θέλουμε να εισάγουμε ή να εξάγουμε. Στην σειριακή εκτέλεση το num_of_threads είναι 1, ενώ στην ταυτόχρονη μπορεί να είναι επιλογή του χρήστη.

```
    long i;
    long long start, end;
    long buffer_size = init->count / init->num_of_threads;
    long curr_range = init->thread_id;

    start = get_ustime_sec();
        for(i = curr_range; i < (curr_range + 1) * buffer_size;
i++) {
            init->curr_element = i;
            write_single_element(init);
```

Οι (write_single_element, και read_single_element) εκτελούν την διαδικασία της προσθήκης ή αφαίρεσης, μία φορά. Ο κώδικας που χρησιμοποιήθηκε ήταν ο ίδιος με το αρχικό.

Τέλος, κάνουμε display τους χρόνους και τα κόστη εκτέλεσης στις συναρτήσεις (display_read_-costs, και display_write_costs).

Εφόσον έχουμε κατηγοριοποιήσει τις λειτουργίες γράφουμε τα τεστ για πολλαπλά threads. Γράφουμε την (_readwrite_test_2_threads) η οποία εκτελεί 1 νήμα για πολλά write και 1 για πολλά read έτσι ώστε να έχουμε ταυτόχρονο read και write.

Παίρνουμε τα read και write initializers αντίστοιχα. Προσέχουμε η βάση να ανοίγει και να κλείνει από ένα μόνο thread.

```
/* Point to the same db, so that we avoid multiple databases */
    read_initializers->db = db_open(DATAS);
    write_initializers->db = read_initializers->db;
```

Μετά, δημιουργούμε τις διαδικασίες των read και write, και κάνουμε join αμέσως μετά. Προσέχουμε οι συναρτήσεις να έχουν ένα γενικευμένο cast σε function pointer που επιστρέφει void* και έχει ένα void* ως παράμετρο. Τέλος παίρνουμε το αποτέλεσμα των execute_and_time* συναρτήσεων (δηλαδή τον τελικό χρόνο εκτέλεσης) και τον αποθηκεύουμε στις διευθύνσεις των 2 double μεταβλητών. Τέλος εκτυπώνουμε το κόστος του read και write καθώς και των συνδυαστικών.

```
/* Try to test for concurrent writes and reads on 2 threads */
    pthread_t t[2];
    pthread_create(&t[0], NULL, (void
*(*)(void*))execute_and_time_write, (void*)write_initializers);
    pthread_create(&t[1], NULL, (void
*(*)(void*))execute_and_time_read, (void*)read_initializers);

    pthread_join(t[0], (void*)&write_cost);
    pthread_join(t[1], (void*)&read_cost);
```

```
/* Display each of the costs */
    display_read_costs(read_cost, count, read_initializers-
>found);
    display_write_costs(write_cost, count);

    printf("Combined cost:%.3f(sec)\n", read_cost + write_cost);

    /* Real cost is calculated monitoring the termination of the
longest thread */
    printf("Total cost:%.3f(sec)\n", (read_cost > write_cost) ?
read_cost : write_cost);
```

Υλοποιούμε την _readwrite_test_X_threads η οποία επιτρέπει την κλήση ταυτόχρονου read και write με δυναμικό αριθμό threads, και ποσοστό τοις εκατό read πράξεων. Όμοια με την _read-write_test_2_threads, παίρνουμε τα initializers όμως θέτουμε τον αριθμό των assigned threads για κάθε διαδικασία ως εξής.

```
/* Offload the processes according to percentage */
    int num_of_read_threads = num_of_threads * read_percentage /
100;
    int num_of_write_threads = num_of_threads -
num_of_read_threads;
    pthread_t *t_read = (pthread_t*)malloc(sizeof(pthread_t) *
num_of_read_threads);
    pthread_t *t_write = (pthread_t*)malloc(sizeof(pthread_t) *
num_of_write_threads);

    /* Set data for threads according to calculated percentage
*/
    r_init->num_of_threads = num_of_read_threads;
    w_init->num_of_threads = num_of_write_threads;
```

Αρχικοποιούμε τα threads στο heap. Ξεκινάμε τα threads αυτή την φορά σε επανάληψη αναλόγως με το ποσό που δόθηκε, και στο τέλος κάνουμε join. Για ξεχωριστή μέτρηση των χρόνων, μία τετριμμένη λύση θα ήταν να προσθέσουμε τον χρόνο εκτέλεσης σε ένα thread safe πεδίο των read_data_t και write_data_t. Όμοια με πριν εκτυπώνουμε τους τελικούς χρόνους.

```
/* Start the cocurrent processing */
    for(wi = 0; wi < num_of_write_threads; wi++) {
        thread_safe_set_w(w_init, "thread_id", (void*)wi);
        pthread_create(&t_write[wi], NULL, (void
*(*)(void*))execute_and_time_write, (void*)w_init);
    }
    for(ri = 0; ri < num_of_read_threads; ri++) {
        thread_safe_set_r(r_init, "thread_id", (void*)ri);
        pthread_create(&t_read[ri], NULL, (void
*(*)(void*))execute_and_time_read, (void*)r_init);
    }

    /* Join all operations */
    for(wi = 0; wi < num_of_write_threads; wi++)
        pthread_join(t_write[wi], (void*)&write_cost);
    for(ri = 0; ri < num_of_read_threads; ri++)
        pthread_join(t_read[ri], (void*)&read_cost);
```

Για να σιγουρέψουμε την συνέπεια στα στοιχεία που μετράμε, για την χρήση και ανανέωση των τιμών χρησιμοποιούμε τις 2 συναρτήσεις (thread_safe_set_w, thread_safe_set_r, thread-_safe_get_w, και thread_safe_get_r). Αυτές παίρνουν σαν παράμετρο ένα struct δεδομένων, το στοιχείο που θέλουμε να πάρουμε ή να ανανεώσουμε εφαρμόζοντας locks.

```
static void thread_safe_set_w(struct write_data_t *init, char
*value, void *new_value) {
    #ifdef BACKGROUND_PUT
        pthread_mutex_lock(&init->self_lock);
    #endif

    if(!strcmp(value, "db"))
        init->db = (DB*)new_value;
    else if(!strcmp(value, "count"))
        init->count = (long)new_value;
    else if(!strcmp(value, "r"))
        init->r = (long)new_value;
    else if(!strcmp(value, "key"))
        init->key = (char*)new_value;
    else if(!strcmp(value, "val"))
        init->val = (char*)new_value;
    else if(!strcmp(value, "sbuf"))
        init->sbuf = (char*)new_value;
    else if(!strcmp(value, "num_of_threads"))
        init->num_of_threads = (long)new_value;
    else if(!strcmp(value, "curr_element"))
        init->curr_element = (long)new_value;
    else if(!strcmp(value, "thread_id"))
        init->thread_id = (long)new_value;

    #ifdef BACKGROUND_PUT
        pthread_mutex_unlock(&init->self_lock);
    #endif
}
```

3. Αποτελέσματα

Τεστάρουμε σε 4 άξονες:

a) Πολλαπλά read (σειριακά και ασύγχρονα).

b) Πολλαπλά write (σειριακά και ασύγχρονα).

c) X read και X write (σειριακά) με X read/write (ασύγχρονα και ταυτόχρονα).

d) X % read και (100-X)% write (σειριακά) και αντίστοιχα read/write.

- ./kiwi-bench write 100000

```
[3211] 30 Mar 19:52:19.765 - file.c:170 Truncating file testdb/si/manifest to 194 bytes
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 0 [  3 files,   1 MiB  ]---
[3211] 30 Mar 19:52:19.766 . sst.c:55 Metadata filenum:399 smallest: key-90574 largest: key-94690
[3211] 30 Mar 19:52:19.766 . sst.c:55 Metadata filenum:400 smallest: key-94691 largest: key-98807
[3211] 30 Mar 19:52:19.766 . sst.c:55 Metadata filenum:401 smallest: key-98808 largest: key-99999
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 1 [  1 files,  19 MiB  ]---
[3211] 30 Mar 19:52:19.766 . sst.c:55 Metadata filenum:398 smallest: key-0 largest: key-99999
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 2 [  1 files,  60 MiB  ]---
[3211] 30 Mar 19:52:19.766 . sst.c:55 Metadata filenum:373 smallest: key-0 largest: key-99999
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 3 [  0 files,   0 bytes]---
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 4 [  0 files,   0 bytes]---
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 5 [  0 files,   0 bytes]---
[3211] 30 Mar 19:52:19.766 . sst.c:51 --- Level 6 [  0 files,   0 bytes]---
[3211] 30 Mar 19:52:19.766 . log.c:50 Removing old log file testdb/si/23.log
[3211] 30 Mar 19:52:19.766 . sst.c:170 Merge successfully completed. Releasing the skiplist
[3211] 30 Mar 19:52:19.766 . skiplist.c:65 SkipList refcount is at 0. Freeing up the structure
[3211] 30 Mar 19:52:19.766 - sst.c:176 Exiting from the merge thread as user requested
[3211] 30 Mar 19:52:19.766 . sst.c:422 Waiting the merger thread
[3211] 30 Mar 19:52:19.766 - file.c:170 Truncating file testdb/si/manifest to 194 bytes
[3211] 30 Mar 19:52:19.772 . log.c:50 Removing old log file testdb/si/24.log
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Write   (done:100000): 0.000020 sec/op; 50000.0 writes/sec(estimated); cost:2.000(sec);
```

- ./kiwi-bench read 100000

```
99985 searching key-99985
99986 searching key-99986
99987 searching key-99987
99988 searching key-99988
99989 searching key-99989
99990 searching key-99990
99991 searching key-99991
99992 searching key-99992
99993 searching key-99993
99994 searching key-99994
99995 searching key-99995
99996 searching key-99996
99997 searching key-99997
99998 searching key-99998
99999 searching key-99999
[2828] 30 Mar 19:38:36.670 . db.c:31 Closing database 0
[2828] 30 Mar 19:38:36.670 . sst.c:415 Sending termination message to the detached thread
[2828] 30 Mar 19:38:36.670 . sst.c:422 Waiting the merger thread
[2828] 30 Mar 19:38:36.670 - sst.c:176 Exiting from the merge thread as user requested
[2828] 30 Mar 19:38:36.670 - file.c:170 Truncating file testdb/si/manifest to 45 bytes
[2828] 30 Mar 19:38:36.680 . log.c:50 Removing old log file testdb/si/0.log
[2828] 30 Mar 19:38:36.680 . skiplist.c:65 SkipList refcount is at 0. Freeing up the structure
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Read    (done:100000, found:100000): 0.000010 sec/op; 100000.0 reads /sec(estimated); cost:1.000(sec)
```

- rm -rf testdb
- ./kiwi-bench readwrite-2 100000

```
[2906] 30 Mar 19:41:09.023 . sst.c:51 --- Level 1 [  2 files,   1 MiB  ]---
[2906] 30 Mar 19:41:09.023 . sst.c:55 Metadata filenum:79 smallest: key-78223 largest: key-82339
[2906] 30 Mar 19:41:09.023 . sst.c:55 Metadata filenum:80 smallest: key-82340 largest: key-86456
[2906] 30 Mar 19:41:09.023 . sst.c:51 --- Level 2 [  1 files,  20 MiB  ]---
[2906] 30 Mar 19:41:09.023 . sst.c:55 Metadata filenum:82 smallest: key-0 largest: key-99999
[2906] 30 Mar 19:41:09.023 . sst.c:51 --- Level 3 [  0 files,   0 bytes]---
[2906] 30 Mar 19:41:09.023 . sst.c:51 --- Level 4 [  0 files,   0 bytes]---
[2906] 30 Mar 19:41:09.023 . sst.c:51 --- Level 5 [  0 files,   0 bytes]---
[2906] 30 Mar 19:41:09.023 . sst.c:51 --- Level 6 [  0 files,   0 bytes]---
[2906] 30 Mar 19:41:09.023 . log.c:50 Removing old log file testdb/si/23.log
[2906] 30 Mar 19:41:09.023 . sst.c:170 Merge successfully completed. Releasing the skiplist
[2906] 30 Mar 19:41:09.023 . skiplist.c:65 SkipList refcount is at 0. Freeing up the structure
[2906] 30 Mar 19:41:09.023 . sst.c:422 Waiting the merger thread
[2906] 30 Mar 19:41:09.023 - sst.c:176 Exiting from the merge thread as user requested
[2906] 30 Mar 19:41:09.024 - file.c:170 Truncating file testdb/si/manifest to 261 bytes
[2906] 30 Mar 19:41:09.034 . log.c:50 Removing old log file testdb/si/24.log
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Read    (done:100000, found:100000): 0.000020 sec/op; 50000.0 reads /sec(estimated); cost:2.000(sec)
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Write   (done:100000): 0.000010 sec/op; 100000.0 writes/sec(estimated); cost:1.000(sec);
Combined cost:3.000(sec)
Total cost:2.000(sec)
```

- ./kiwi-bench read 100000

```
99998 searching key-99998
99999 searching key-99999
[2929] 30 Mar 19:42:13.452 . db.c:31 Closing database 0
[2929] 30 Mar 19:42:13.452 . sst.c:415 Sending termination message to the detached thread
[2929] 30 Mar 19:42:13.452 . sst.c:422 Waiting the merger thread
[2929] 30 Mar 19:42:13.452 - sst.c:176 Exiting from the merge thread as user requested
[2929] 30 Mar 19:42:13.453 - file.c:170 Truncating file testdb/si/manifest to 261 bytes
[2929] 30 Mar 19:42:13.464 . log.c:50 Removing old log file testdb/si/0.log
[2929] 30 Mar 19:42:13.464 . skiplist.c:65 SkipList refcount is at 0. Freeing up the structure
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Read    (done:100000, found:100000): 0.000010 sec/op; 100000.0 reads /sec(estimated); cost:1.000(sec)
```

- rm -rf testdb
- ./kiwi-bench readwrite-x 4000 2 # 4000 elements on 2 read and 2 write threads

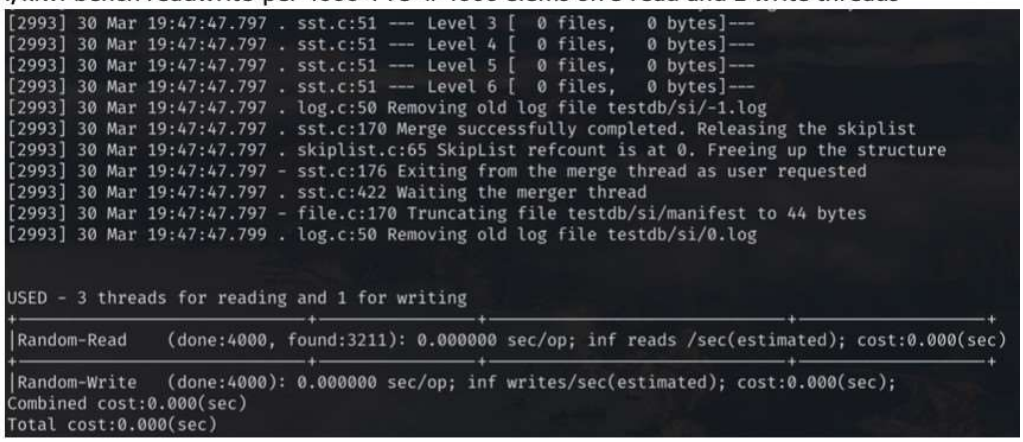
```
[2955] 30 Mar 19:43:19.355 . sst.c:55 Metadata filenum:4 smallest: key-3840 largest: key-3999
[2955] 30 Mar 19:43:19.355 . sst.c:51 --- Level 1 [  1 files, 856 KiB  ]---
[2955] 30 Mar 19:43:19.355 . sst.c:55 Metadata filenum:1 smallest:  largest: key-999
[2955] 30 Mar 19:43:19.355 . sst.c:51 --- Level 2 [  1 files, 479 KiB  ]---
[2955] 30 Mar 19:43:19.355 . sst.c:55 Metadata filenum:0 smallest: key-1 largest: key-999
[2955] 30 Mar 19:43:19.355 . sst.c:51 --- Level 3 [  0 files,   0 bytes]---
[2955] 30 Mar 19:43:19.355 . sst.c:51 --- Level 4 [  0 files,   0 bytes]---
[2955] 30 Mar 19:43:19.355 . sst.c:51 --- Level 5 [  0 files,   0 bytes]---
[2955] 30 Mar 19:43:19.355 . sst.c:51 --- Level 6 [  0 files,   0 bytes]---
[2955] 30 Mar 19:43:19.355 . log.c:50 Removing old log file testdb/si/2.log
[2955] 30 Mar 19:43:19.355 . sst.c:170 Merge successfully completed. Releasing the skiplist
[2955] 30 Mar 19:43:19.355 . skiplist.c:65 SkipList refcount is at 0. Freeing up the structure
[2955] 30 Mar 19:43:19.355 - sst.c:176 Exiting from the merge thread as user requested
[2955] 30 Mar 19:43:19.355 - file.c:170 Truncating file testdb/si/manifest to 172 bytes
[2955] 30 Mar 19:43:19.357 . log.c:50 Removing old log file testdb/si/3.log


USED - 2 threads for reading and 2 for writing
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Read    (done:4000, found:3881): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Write   (done:4000): 0.000000 sec/op; inf writes/sec(estimated); cost:0.000(sec);
Combined cost:0.000(sec)
Total cost:0.000(sec)
```

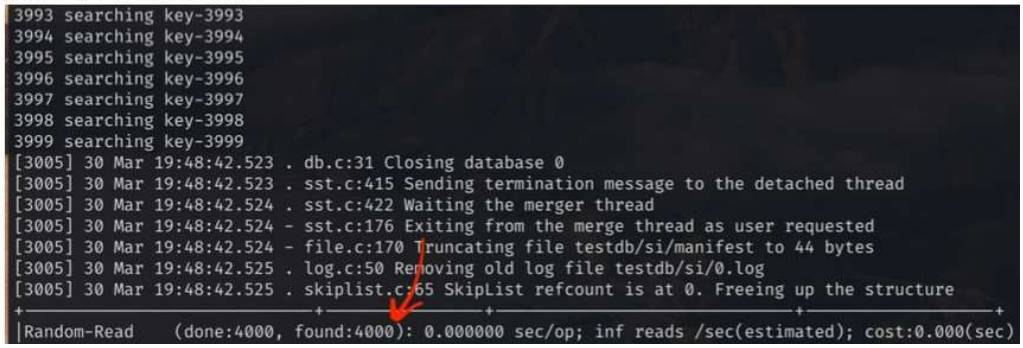
- ./kiwi-bench read 4000

```
3998 searching key-3998
3999 searching key-3999
[2963] 30 Mar 19:44:15.773 . db.c:31 Closing database 0
[2963] 30 Mar 19:44:15.773 . sst.c:415 Sending termination message to the detached thread
[2963] 30 Mar 19:44:15.773 . sst.c:422 Waiting the merger thread
[2963] 30 Mar 19:44:15.773 - sst.c:176 Exiting from the merge thread as user requested
[2963] 30 Mar 19:44:15.774 - file.c:170 Truncating file testdb/si/manifest to 172 bytes
[2963] 30 Mar 19:44:15.776 . log.c:50 Removing old log file testdb/si/0.log
[2963] 30 Mar 19:44:15.776 . skiplist.c:65 SkipList refcount is at 0. Freeing up the structure
+-------------------------------+-----------------+-------------------------------+-------------------+
|Random-Read    (done:4000, found:3942): 0.000000 sec/op; inf reads /sec(estimated); cost:0.000(sec)
```

- rm -rf testdb
- ./kiwi-bench readwrite-per 4000 4 75 # 4000 elems on 3 read and 1 write threads



- ./kiwi-bench read 4000



Στο readwrite-2 παρατηρούμε ότι το read αμέσως μετά βρίσκει όλα τα στοιχεία, άρα υπάρχει συνέπεια στις δομές. Ωστόσο στο readrwite-x και readwrite-per, μερικές φορές δεν βρίσκονται όλα τα στοιχεία, καθώς επίσης έχουμε segfaults μετα το πρώτο merge, εφόσον υπάρχει overlapping τιμών. Το πρόβλημα μπορούμε να το αντιμετωπίσουμε εφαρμόζοντας την απλή μέθοδο του κλειδώματος της db_add και db_get ολοκληρωτικά (τα έχουμε σε σχόλια). Έτσι μπορούμε να χρησιμοποιήσουμε και παραπάνω από 4000 τιμές.

```
int db_get(DB* self, Variant* key, Variant* value)
{
// #if defined(BACKGROUND_GET) && defined(GLOBAL_LOCK)
//     pthread_mutex_lock(&self->get_mutex);
// #endif

    if (memtable_get(self->memtable->list, key, value) == 1) {
        #if defined(BACKGROUND_GET) && defined(GLOBAL_LOCK)
            pthread_mutex_unlock(&self->get_mutex);
        #endif
        return 1;
```

```
    }

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

// #if defined(BACKGROUND_GET) && defined(GLOBAL_LOCK)
//     pthread_mutex_unlock(&self->get_mutex);
// #endif

    return res;
}
```

```
int db_add(DB* self, Variant* key, Variant* value)
{
// #if defined(BACKGROUND_PUT) && defined(GLOBAL_LOCK)
//     pthread_mutex_lock(&self->put_mutex);
// #endif

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

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

// #if defined(BACKGROUND_PUT) && defined(GLOBAL_LOCK)
//     pthread_mutex_unlock(&self->put_mutex);
// #endif

    return res;
}
```

4. Προβλήματα

Σπάνια και όχι επαναλαμβανόμενα έχουμε segfaults στην db_add όπου με διαγραφή του testdb και εκτέλεση ξανά δουλεύει χωρίς λάθη. Το γεγονός πως τα segfaults και τα double frees είναι τυχαία και ότι αυξάνονται όσο περισσότερα στοιχεία προθέτουμε σημαίνει ότι δεν έχουμε καλύψει αναλυτικά όλες τις κρίσιμες περιοχές ωστόσο φαίνεται να υπάρχει συνέπεια μετά τις προσθήκες καθώς δεν έχουμε πετύχει κάποιο false positive ή false negative. Για να αντιμετωπίσουμε το πρόβλημα αφαιρέσαμε μία κλήση της log_free στην db_close η οποία μετά από έλεγχο σε debugger και profiler είδαμε πως εκτελεί double_free. Προφανώς αυτή δεν είναι η επιθυμητή λύση, όμως δεν μπορέσαμε να λύσουμε διαφορετικά το πρόβλημα.

```
void db_close(DB *self)
{
    INFO("Closing database %d", self->memtable->add_count);

    if (self->memtable->list->count > 0)
    {
        sst_merge(self->sst, self->memtable);
        skiplist_release(self->memtable->list);
        self->memtable->list = NULL;
    }

    sst_free(self->sst);
    log_remove(self->memtable->log, self->memtable->lsn);
    // log_free(self->memtable->log);
    memtable_free(self->memtable);
    free(self);
}
```

Στην write_single_element συναντήσαμε bus error στην εκτέλεση του snprintf της γραμμής 289 του kiwi.c. Αποφασίσαμε να βάλουμε μία τυχαία τιμή στο sv.mem της 295 για να αποφύγουμε το λάθος, επειδή φαίνεται το snprintf να ποδοπατάει κομμάτια μνήμης που δεν επιτρέπεται, και αφού στην αναζήτηση κοιτάμε το key έτσι κι αλλιώς. Φυσικά αυτό βάζει λάθος τιμές στην βάση δεδομένων και δεν θα ήταν σε καμία περίπτωση μία τελική λύση.

```
/* TODO -> BUS ERROR */
    // snprintf(init->val, VSIZE, "val-%ld", init->i);

    sk.length = KSIZE;
    sk.mem = init->key;
    sv.length = VSIZE;

    // sv.mem = init->val;
    char *sbmemstring = (char*)malloc(32);
    sbmemstring = "value";
    sv.mem = sbmemstring;
```