

Εργαστήριο 1: Υλοποίηση πολυνηματικής λειτουργίας σε μηχανή αποθήκευσης δεδομένων

Ομάδα: Θεοφάνης Τομπόλης, 4855

Αθανάσιος Φυτιλής, 5381

Μαρίνος Αριστείδου, 5397

A. Εισαγωγή

1. Γενικά

Η αποθήκευση και ανάκτηση δεδομένων αποτελεί θεμελιώδη λειτουργία για κάθε υπολογιστικό σύστημα που διαχειρίζεται πληροφορία, ιδιαίτερα σε περιβάλλοντα με αυξημένες απαιτήσεις απόδοσης, αξιοπιστίας και παραλληλίας. Οι σύγχρονες εφαρμογές – από βάσεις δεδομένων έως κατανεμημένες πλατφόρμες cloud – απαιτούν μηχανές αποθήκευσης ικανές να διαχειρίζονται μεγάλα σύνολα δεδομένων, υποστηρίζοντας παράλληλη πρόσβαση με σταθερή και αποδοτική συμπεριφορά.

Η συγκεκριμένη μηχανή αποθήκευσης που μελετάται βασίζεται στην αρχιτεκτονική του **Log-Structured Merge-tree (LSM-tree)**, μια ευρέως διαδεδομένη προσέγγιση για τη βελτιστοποίηση των λειτουργιών εγγραφής και ανάγνωσης μέσω συγχωνεύσεων επιπέδων και διατήρησης δεδομένων σε διακριτά segments. Μέσω της διεπαφής της, παρέχει βασικές λειτουργίες διαχείρισης δεδομένων (add, get) σε μορφή ζευγών κλειδιού-τιμής, ενώ ενσωματώνει μηχανισμούς για την αποδοτική ανάκτηση και περιοδική συγχώνευση των αποθηκευμένων δεδομένων.

Παρά τις ενσωματωμένες βελτιστοποιήσεις, οι περισσότερες τέτοιες υλοποιήσεις δεν είναι πλήρως προσαρμοσμένες για ταυτόχρονη εκτέλεση λειτουργιών από πολλαπλά νήματα. Η απουσία κατάλληλου μηχανισμού συγχρονισμού συχνά οδηγεί σε φαινόμενα όπως συνθήκες ανταγωνισμού, απώλεια δεδομένων ή και πτώση απόδοσης, καθιστώντας την ανάγκη για πολυνηματική υποστήριξη ιδιαίτερα σημαντική.

2. Στόχος Υλοποίησης

Η παρούσα εργασία επικεντρώνεται στην ενίσχυση της μηχανής αποθήκευσης με πολυνηματική λειτουργικότητα, επιτρέποντας την ταυτόχρονη εκτέλεση λειτουργιών από πολλαπλά νήματα με τρόπο αποδοτικό και ασφαλή. Βασικό ζητούμενο είναι η σχεδίαση και υλοποίηση ενός μηχανισμού συγχρονισμού που θα υποστηρίζει πολλαπλούς αναγνώστες

ΜΥΥ601 Λειτουργικά Συστήματα

και **έναν συγγραφέα ταυτόχρονα** (readers-writer model), χωρίς να τίθεται σε κίνδυνο η συνοχή και η ορθότητα των δεδομένων.

Για την υλοποίηση αυτής της δυνατότητας, χρησιμοποιούνται τεχνικές όπως **κλειδώματα** (**mutex locks**) και **μεταβλητές συνθήκης** (**condition variables**), επιτρέποντας τη ρύθμιση της πρόσβασης σε κοινόχρηστους πόρους και την αποφυγή φαινομένων όπως αδιέξοδα (deadlocks) ή αστοχίες συγχρονισμού. Επιπλέον, ενσωματώθηκαν μηχανισμοί για τον δίκαιο διαμοιρασμό των εργασιών στα νήματα, διατηρώντας την ισορροπία μεταξύ απόδοσης και αξιοπιστίας.

Τέλος, πραγματοποιήθηκε **δοκιμαστική αξιολόγηση** (benchmarking) της απόδοσης της μηχανής, εστιάζοντας σε μετρικές όπως **χρόνος απόκρισης** και **ρυθμός εξυπηρέτησης αιτημάτων** (requests per second), υπό διαφορετικά επίπεδα ταυτόχρονης πρόσβασης. Μέσα από αυτήν την ανάλυση, αναδεικνύονται τα οφέλη και οι περιορισμοί της προσέγγισης, συμβάλλοντας στην κατανόηση της συμπεριφοράς της μηχανής υπό συνθήκες φόρτου και υψηλής ταυτόχρονης δραστηριότητας.

3. Λεξιλόγιο και Παρατηρήσεις

- Δοκιμαστική Αξιολόγηση = Benchmark = Πρόγραμμα αξιολόγησης
- Λειτουργίες = Εργασίες / Συναρτήσεις
- Ιδιαίτερη έμφαση δίνεται στις **προδιαγραφές συστήματος**, καθώς αυτές επηρεάζουν άμεσα τη μετρούμενη απόδοση της υλοποίησης.

B. Κύριες Τροποποιήσεις στον Κώδικα

Αρχείο 'bench.c':

Mix Thread

Η συνάρτηση `mix_thread` υλοποιεί τη βασική λογική που εκτελεί κάθε νήμα στο πλαίσιο του μεικτού benchmarking. Κάθε νήμα πραγματοποιεί έναν προκαθορισμένο αριθμό λειτουργιών `read` και `write`, σύμφωνα με το ποσοστό εγγραφών που έχει οριστεί από τον χρήστη. Ανάλογα με τις παραμέτρους, τα κλειδιά μπορούν να είναι είτε τυχαία είτε σειριακά. Για κάθε λειτουργία μετρέται ο χρόνος εκτέλεσης σε μικροδευτερόλεπτα και συγκεντρώνονται στατιστικά, όπως πλήθος reads/writes και συνολικός χρόνος λειτουργίας του thread. Η υλοποίηση επιτρέπει λεπτομερή καταγραφή επιδόσεων και προσομοιώνει ρεαλιστικά σενάρια φόρτου για τη μηχανή αποθήκευσης.

```
void* mix_thread(void* arg) {
    thread_args* args = (thread_args*) arg;
    char key[KSIZE + 1];
    char val[VSIZE + 1]; // Only needed for writes
    Variant k, v;
```

ΜΥΥ601 Λειτουργικά Συστήματα

```
k.mem = key; // Point k.mem once
v.mem = val; // Point v.mem once
k.length = KSIZE;
v.length = VSIZE; // VSIZE might be optimized away by compiler if not used

// Initialize results
args->reads_done = 0;
args->writes_done = 0;
args->read_time_us = 0;
args->write_time_us = 0;

uint64_t thread_start_us = get_monotonic_us();

for (int i = 0; i < args->ops_count; i++) {
    // Determine operation type
    int do_write = (rand() % 100) < args->write_ratio_percent;

    // Generate Key
    if (args->use_random_keys) {
        _random_key(key, KSIZE);
    } else {
        // Consistent key generation for sequential benchmarks if needed
        // Note: Ensure this doesn't collide heavily with few threads/ops
        // Consider a larger range or better distribution for non-random
        sprintf(key, KSIZE + 1, "key-%0*d", KSIZE - 4, i + args->thread_id * args->ops_count); // Padded key
    }
    // k.length = KSIZE; // Already set

    if (do_write) {
        // Generate Value (only when writing)
        // Simple value generation - could be made more complex if needed
        sprintf(val, VSIZE + 1, "val-%d", i); // Keep null termination for safety
    }
    // v.length = VSIZE; // Already set

    uint64_t op_start_us = get_monotonic_us();
    db_add(db, &k, &v); // <<< --- TIME THIS --- >>>
    uint64_t op_end_us = get_monotonic_us();

    args->writes_done++;
    args->write_time_us += (op_end_us - op_start_us);

} else { // Read operation
    uint64_t op_start_us = get_monotonic_us();
    // Result of db_get not used, common in benchmarks (check existence/latency)
    db_get(db, &k, &v); // <<< --- TIME THIS --- >>>
    uint64_t op_end_us = get_monotonic_us();
    // TODO: Optionally check the return value of db_get if needed

    args->reads_done++;
    args->read_time_us += (op_end_us - op_start_us);
}

uint64_t thread_end_us = get_monotonic_us();
args->total_time_ms = (thread_end_us - thread_start_us) / 1000; // Store thread wall
time in ms

return NULL;
}
```

```
thread_args* args = (thread_args*) arg;
```

- ⇒ Μετατροπή του void pointer σε κατάλληρη δομή δεδομένων που περιέχει τις παραμέτρους του νήματος.

```
char key[KSIZE + 1]; char val[VSIZE + 1];
```

- ⇒ Δημιουργία buffers για key και value που θα χρησιμοποιηθούν σε read/write λειτουργίες.

```
k.mem = key; v.mem = val;
```

- ⇒ Αρχικοποίηση των Variant δομών για να δείχνουν στα αντίστοιχα buffers.

```
args->reads_done = 0; args->writes_done = 0;
```

- ⇒ Μηδενισμός counters που μετρούν read και write επιτυχίες για το νήμα.

```
int do_write = (rand() % 100) < args->write_ratio_percent;
```

- ⇒ Τυχαία επιλογή αν θα εκτελεστεί write ή read, με βάση το ποσοστό εγγραφών.

```
_random_key(key, KSIZE);
```

- ⇒ Γεννάει τυχαίο κλειδί αν έχει ζητηθεί random mode από τον χρήστη.

```
snprintf(key, KSIZE + 1, "key-%0*d", KSIZE - 4, i + args->thread_id * args->ops_count);
```

- ⇒ Γεννάει σειριακό κλειδί, διαφορετικό ανά νήμα και λειτουργία (αν δεν είναι random mode).

```
snprintf(val, VSIZE + 1, "val-%d", i);
```

- ⇒ Γεννάει την τιμή που θα αποθηκευτεί σε περίπτωση write.

```
uint64_t op_start_us = get_monotonic_us(); ... get_monotonic_us();
```

⇒ Χρήση χρονόμετρου ακριβείας για μέτρηση διάρκειας read ή write λειτουργίας.

```
args->write_time_us += (op_end_us - op_start_us);
```

⇒ Καταγραφή συνολικού χρόνου που δαπανήθηκε σε write λειτουργίες από το νήμα.

```
args->total_time_ms = (thread_end_us - thread_start_us) / 1000;
```

⇒ Αποθήκευση συνολικού χρόνου εκτέλεσης του νήματος (σε ms) για αναφορά.

Run Mix

Η συνάρτηση `run_mix` είναι υπεύθυνη για την **εκτέλεση πολυνηματικού benchmarking** με μεικτές λειτουργίες `read` και `write`. Λαμβάνει από τον χρήστη τον αριθμό συνολικών λειτουργιών, το ποσοστό εγγραφών, τον αριθμό νημάτων και το αν θα χρησιμοποιηθούν τυχαία κλειδιά. Δημιουργεί και εκκινεί threads, συλλέγει στατιστικά απόδοσης και υπολογίζει μετρικές όπως **latency**, **throughput** και **χρόνος τοίχου (wall time)**.

```
void* mix_thread(void* arg) {
    thread_args* args = (thread_args*) arg;
    char key[KSIZE + 1];
    char val[VSIZE + 1]; // Only needed for writes
    Variant k, v;
    k.mem = key; // Point k.mem once
    v.mem = val; // Point v.mem once
    k.length = KSIZE;
    v.length = VSIZE; // VSIZE might be optimized away by compiler if not used

    // Initialize results
    args->reads_done = 0;
    args->writes_done = 0;
    args->read_time_us = 0;
    args->write_time_us = 0;

    uint64_t thread_start_us = get_monotonic_us();

    for (int i = 0; i < args->ops_count; i++) {
        // Determine operation type
        int do_write = (rand() % 100) < args->write_ratio_percent;
```

ΜΥΥ601 Λειτουργικά Συστήματα

```
// Generate Key
if (args->use_random_keys) {
    _random_key(key, KSIZE);
} else {
    // Consistent key generation for sequential benchmarks if needed
    // Note: Ensure this doesn't collide heavily with few threads/ops
    // Consider a larger range or better distribution for non-random
    sprintf(key, KSIZE + 1, "key-%0*d", KSIZE - 4, i + args->thread_id * args-
>ops_count); // Padded key
}
// k.length = KSIZE; // Already set

if (do_write) {
    // Generate Value (only when writing)
    // Simple value generation - could be made more complex if needed
    sprintf(val, VSIZE + 1, "val-%d", i); // Keep null termination for safety
}

if needed
    // v.length = VSIZE; // Already set

    uint64_t op_start_us = get_monotonic_us();
    db_add(db, &k, &v); // <<< --- TIME THIS --- >>>
    uint64_t op_end_us = get_monotonic_us();

    args->writes_done++;
    args->write_time_us += (op_end_us - op_start_us);

} else { // Read operation
    uint64_t op_start_us = get_monotonic_us();
    // Result of db_get not used, common in benchmarks (check existence/latency)
    db_get(db, &k, &v); // <<< --- TIME THIS --- >>>
    uint64_t op_end_us = get_monotonic_us();
    // TODO: Optionally check the return value of db_get if needed

    args->reads_done++;
    args->read_time_us += (op_end_us - op_start_us);
}
}

uint64_t thread_end_us = get_monotonic_us();
args->total_time_ms = (thread_end_us - thread_start_us) / 1000; // Store thread wall
time in ms

return NULL;
}

// --- Main Benchmark Runner for MIX mode ---
void run_mix(int total_ops_requested, int write_ratio_percent, int num_threads, int
use_random_keys) {
    pthread_t tids[num_threads];
    thread_args args[num_threads];

    // Calculate ops per thread, handle uneven division
    int ops_per_thread = total_ops_requested / num_threads;
    int remainder_ops = total_ops_requested % num_threads;
    int total_ops_actual = 0; // Sum of ops actually assigned

    printf("Starting benchmark...\n");
    printf("Requested Total Ops: %d\n", total_ops_requested);
    printf("Threads: %d, Write Ratio: %d%%\n", num_threads, write_ratio_percent);
    printf("Random Keys: %s\n", use_random_keys ? "Yes" : "No");
}
```

ΜΥΥ601 Λειτουργικά Συστήματα

```
printf("-----\n");

uint64_t start_us = get_monotonic_us();

db = db_open("testdb");
if (!db) {
    fprintf(stderr, "ERROR: Failed to open database 'testdb'\n");
    exit(EXIT_FAILURE);
}

// Launch threads
for (int i = 0; i < num_threads; i++) {
    args[i].thread_id = i;
    args[i].ops_count = ops_per_thread + (i < remainder_ops ? 1 : 0); // Distribute
    remainder
    args[i].write_ratio_percent = write_ratio_percent;
    args[i].use_random_keys = use_random_keys;
    total_ops_actual += args[i].ops_count; // Track actual ops assigned

    if (pthread_create(&tids[i], NULL, mix_thread, &args[i]) != 0) {
        perror("pthread_create");
        // Consider cleanup / joining already created threads before exiting
        exit(EXIT_FAILURE);
    }
}

// Wait for threads to complete and Aggregate results
total_reads_global = 0;
total_writes_global = 0;
total_read_time_us = 0;
total_write_time_us = 0;
uint64_t max_thread_time_ms = 0; // Time taken by the slowest thread

for (int i = 0; i < num_threads; i++) {
    if (pthread_join(tids[i], NULL) != 0) {
        perror("pthread_join");
        // Consider how to handle join errors
    }
    // Aggregate results *after* thread has finished
    total_reads_global += args[i].reads_done;
    total_writes_global += args[i].writes_done;
    total_read_time_us += args[i].read_time_us;
    total_write_time_us += args[i].write_time_us;
    if (args[i].total_time_ms > max_thread_time_ms) {
        max_thread_time_ms = args[i].total_time_ms;
    }
}

uint64_t end_us = get_monotonic_us();
db_close(db); // Close DB after all threads are done

// --- Calculate Summary Statistics ---
double total_duration_sec = (end_us - start_us) / 1000000.0;
uint64_t total_ops_done = total_reads_global + total_writes_global;
// Ensure total_ops_actual matches total_ops_done if threads complete successfully
if (total_ops_done != (uint64_t)total_ops_actual) {
    fprintf(stderr, "Warning: Mismatch between assigned ops (%d) and completed ops
(%" PRIu64 ")\n",
            total_ops_actual, total_ops_done);
```

```

    }

    double ops_per_sec = (total_duration_sec > 0) ? (total_ops_done /
total_duration_sec) : 0;

    // Calculate average latencies in microseconds
    double avg_read_latency_us = (total_reads_global > 0) ? ((double)total_read_time_us /
total_reads_global) : 0;
    double avg_write_latency_us = (total_writes_global > 0) ?
((double)total_write_time_us / total_writes_global) : 0;
    uint64_t total_op_time_us = total_read_time_us + total_write_time_us;
    double avg_op_latency_us = (total_ops_done > 0) ? ((double)total_op_time_us /
total_ops_done) : 0;

    // --- Print Summary Output ---
    printf("+-+-----+-----+\n");
    printf(" | Overall Statistics | |\n");
    printf("+-+-----+-----+\n");
    printf(" | Wall Clock Time | %10.3f sec | \n",
total_duration_sec);
    printf(" | Total Operations Completed | %10" PRIu64 " ops | \n",
total_ops_done);
    printf(" | Reads | %10" PRIu64 " ops | \n",
total_reads_global);
    printf(" | Writes | %10" PRIu64 " ops | \n",
total_writes_global);
    printf(" | Throughput | %10.1f ops/sec | \n",
ops_per_sec);
    printf("+-+-----+-----+\n");
    printf(" | Average Latencies | |\n");
    printf("+-+-----+-----+\n");
    printf(" | Average Operation Latency | %10.1f us | \n",
avg_op_latency_us);
    printf(" | Average Read Latency | %10.1f us | \n",
avg_read_latency_us);
    printf(" | Average Write Latency | %10.1f us | \n",
avg_write_latency_us);
    printf("+-+-----+-----+\n");
    // Optionally print total CPU time spent inside DB ops (convert us to sec)
    printf(" | Total DB Ops CPU Time (Sum) | %10.3f sec | \n",
total_op_time_us / 1000000.0);
    printf(" | Max Thread Wall Time | %10" PRIu64 " ms | \n",
max_thread_time_ms);
    printf("+-+-----+-----+\n");

    // Optional: Print per-thread summary (can be verbose)

    printf("\nPer-Thread Details:\n");
    printf("-----\n");
    printf(" | Thread | Ops Done | Reads | Writes | Read Time (us) | Write Time (us) | \n");
    printf(" | -----|-----|-----|-----|-----|-----|-----|\n");
    for(int i = 0; i < num_threads; i++) {
        printf(" | %6d | %-8" PRIu64 " | %-6" PRIu64 " | %-6" PRIu64 " | %14" PRIu64 " |
%-15" PRIu64 " | %-14" PRIu64 " | \n",
            args[i].thread_id,
            args[i].reads_done + args[i].writes_done,
            args[i].reads_done,
            args[i].writes_done,
            args[i].read_time_us,
            args[i].write_time_us);
    }
}

```

```
        args[i].writes_done,
        args[i].read_time_us,
        args[i].write_time_us,
        args[i].total_time_ms);
    }
    printf("-----\n");
}
}
```

- `pthread_t tids[threads];`: Δημιουργεί πίνακα από `pthread_t` handles για τα νήματα.
- `thread_args args[threads];`: Ορίζει πίνακα δομών για τα ορίσματα κάθε νήματος.
- `int ops_per_thread = total_ops / threads;`: Μοιράζει ισομερώς τις λειτουργίες σε όλα τα νήματα.

```
total_reads_global += args[i].reads_done;
total_writes_global += args[i].writes_done;
```

- Μετά το `join`, προσθέτει τα αποτελέσματα κάθε νήματος στο συνολικό άθροισμα.
 - `pthread_create(&tids[i], NULL, mix_thread, &args[i]);`: Δημιουργεί νήμα για εκτέλεση της `mix_thread`.
 - `pthread_join(tids[i], NULL);`: Περιμένει να ολοκληρωθεί κάθε νήμα πριν συνεχίσει.
 - `db_close(db);`: Κλείνει τη βάση δεδομένων μετά την ολοκλήρωση όλων των νήματων.
 - `printf(" | Total Operations Completed | ... |");`
`printf(" | Average Write Latency | ... |");`
- ⇒ Εμφανίζει τελικό συνοπτικό απολογισμό της εκτέλεσης με όλες τις βασικές μετρικές.

```
printf(" | %-6d | %-8" PRIu64 " | ... |\\n", args[i].thread_id, ...);
```

- ⇒ Αναλυτική εμφάνιση των αποτελεσμάτων κάθε νήματος ξεχωριστά (πολύ χρήσιμο για debugging και ανάλυση φόρτου).

Main

Η `main` τροποποιήθηκε ώστε να αναγνωρίζει την επιλογή "mix" μέσω της γραμμής εντολών

ΜΥΥ601 Λειτουργικά Συστήματα

και να διαχειρίζεται παραμέτρους όπως αριθμό λειτουργιών, ποσοστό εγγραφών, πλήθος νημάτων και τύπο κλειδιών (random ή σταθερά). Με αυτό τον τρόπο, ο χρήστης έχει τη δυνατότητα να ελέγχει με ακρίβεια τις συνθήκες των δοκιμών και να προσομοιώνει διάφορα σενάρια χρήσης

```
if (argc >= 2 && strcmp(argv[1], "mix") == 0) {  
• if (argc >= 2 && strcmp(argv[1], "mix") == 0): Ελέγχει αν ζητήθηκε η λειτουργία mix από τον χρήστη.
```

```
long long total_ops_ll = atoll(argv[2]); // Use long long for potentially large op counts  
int write_ratio_percent = atoi(argv[3]);  
int threads = atoi(argv[4]);  
int random_keys = atoi(argv[5]);
```

• long long total_ops_ll = atoll(argv[2]); Ανάγνωση αριθμού total ops ως long long.

- int write_ratio_percent = atoi(argv[3]); Ποσοστό εγγραφών.
- int threads = atoi(argv[4]); Αριθμός νημάτων
- int random_keys = atoi(argv[5]); Boolean για χρήση τυχαίων κλειδιών

```
if (total_ops_ll <= 0 || write_ratio_percent < 0 || write_ratio_percent > 100 || threads <= 0 || (random_keys != 0 && random_keys != 1)) {  
    fprintf(stderr, "Error: Invalid arguments.\n");  
    fprintf(stderr, "Usage: ./kiwi-bench mix <total_ops> <write_%> <threads> <random(0|1)>\n");  
    exit(EXIT_FAILURE);
```

Έλεγχοι εγκυρότητας για τις παραμέτρους. Αποφυγή μηδενικών/αρνητικών/λανθασμένων τιμών

```
if (total_ops_ll > 0 && total_ops_ll < threads) {  
    fprintf(stderr, "Warning: Total operations (%lld) is less than thread count (%d). Adjusting threads to %lld.\n",  
            total_ops_ll, threads, total_ops_ll);  
    threads = (int)total_ops_ll;
```

Προσαρμογή αριθμού νημάτων αν ξεπερνά τα total ops. Βελτιστοποίηση

```
int total_ops_int = (int)total_ops_ll; // Cast for functions expecting int, check for overflow if needed  
• int ops = atoi(argv[2]): Μετατρέπει τον αριθμό λειτουργιών από string σε int.
```

```
print_header(total_ops_int);  
_print_environment();  
run_mix(total_ops_int, write_ratio_percent, threads, random_keys);  
_print_header(total_ops_int); Εμφάνιση στατιστικών header
```

print_environment(); Εμφάνιση περιβάλλοντος CPU

run_mix(total_ops_int, write_ratio_percent, threads, random_keys); Κλήση της νέας mix συνάρτησης

Print Environment

Η συνάρτηση `_print_environment` βελτιώθηκε ώστε να διαβάζει με πιο ασφαλή τρόπο πληροφορίες για το CPU και την cache του συστήματος. Αυτό μας βοηθάει να γνωρίζουμε σε τι περιβάλλον εκτελείται η δοκιμή, κάτι κρίσιμο για τη σύγκριση αποτελεσμάτων μεταξύ διαφορετικών μηχανών.

```
int core_count = 0; // Count physical processors listed  
int core_count = 0; καταμέτρηση φυσικών επεξεργαστών
```

```
char* start = line; while (isspace(*start)) start++;  
    char* end = start + strlen(start) - 1; while (end > start && isspace(*end))  
end--; *(end + 1) = '\0';
```

Trim whitespaces από κάθε γραμμή. Καλύτερη ανθεκτικότητα στην επεξεργασία του `/proc/cpuinfo`

```
if (strcmp(start, "processor", 9) == 0) { // Count lines starting with "processor"  
    core_count++;  
} else if (strcmp(start, "model name", 10) == 0 && cpu_type[0] == '\0') {  
// Take first model name
```

Ανίχνευση γραμμών με processor, model name, cache size για πιο σαφές parsing.

- `strncpy(cpu_type, colon + 2, sizeof(cpu_type) - 1);`: Αντιγράφει το περιεχόμενο της CPU μετά το ":".

```
char* colon = strchr(start, ':');  
if (colon) {  
    char* model = colon + 1; while (isspace(*model)) model++; // Skip  
space after colon  
    strncpy(cpu_type, model, sizeof(cpu_type) - 1);
```

Απόσπαση μοντέλου CPU. Η εντολή `char* colon = strchr(start, ':');` εντοπίζει τη θέση του χαρακτήρα : μέσα σε μια γραμμή κειμένου. Χρησιμοποιείται για να διαχωριστεί το key από το value κατά το parsing πληροφοριών, π.χ. από το `/proc/cpuinfo`.

```
} else if (strcmp(start, "cache size", 10) == 0 && cache_size[0] == '\0') { // Take  
first cache size  
    char* colon = strchr(start, ':');  
    if (colon) {  
        char* cache = colon + 1; while (isspace(*cache)) cache++; // Skip  
space  
        strncpy(cache_size, cache, sizeof(cache_size) - 1);
```

Απόσπαση μεγέθους cache,όμοιο με παραπάνω.

```
num_cpus = core_count;  
num_cpus = core_count; Χρήση του πλήθους processor
```

```
printf("CPU:\t%d * %s\n", num_cpus, cpu_type[0] ? cpu_type : "Unknown");  
printf("CPUCache:\t%s\n", cache_size[0] ? cache_size : "Unknown");
```

Εκτύπωση τύπου CPU.

- `strstr(line, "model name")`: Αναγνωρίζει τη γραμμή που περιέχει τον τύπο CPU.

```
while (fgets(line, sizeof(line), ccpuinfo) != NULL)
```

- `fgets(line, ...)` μέσα σε loop: Διαβάζει γραμμή-γραμμή το αρχείο /proc/cpuinfo.

```
fclose(ccpuinfo);
```

- `fclose(ccpuinfo);`: Κλείνει το αρχείο για να αποφευχθεί memory leak.

Helpers & Globals

Η δομή `thread_args` περιέχει τις παραμέτρους και τα στατιστικά που χρειάζεται κάθε νήμα κατά την εκτέλεση του benchmarking. Αποθηκεύει τόσο τις ρυθμίσεις (όπως αριθμός λειτουργιών και ποσοστό εγγραφών), όσο και τα αποτελέσματα (π.χ. χρόνος read/write και πλήθος λειτουργιών), επιτρέποντας την ανεξάρτητη λειτουργία κάθε νήματος.

```
typedef struct {

    int thread_id;
    int ops_count; // Number of operations this thread should perform
    int write_ratio_percent; // Write percentage (0-100)
    int use_random_keys;

    // Per-thread results
    uint64_t reads_done;
    uint64_t writes_done;
    uint64_t read_time_us; // Total microseconds spent in db_get
    uint64_t write_time_us; // Total microseconds spent in db_add
    uint64_t total_time_ms; // Wall clock time for this thread
} thread_args;
```

Οι μεταβλητές `total_reads_global`, `total_writes_global`, `total_read_time_us` και `total_write_time_us` χρησιμοποιούνται για τη συγκέντρωση συνολικών στατιστικών από όλα τα threads μετά την ολοκλήρωση τους. Επειδή η πρόσβαση γίνεται μόνο αφού έχουν τερματίσει τα νήματα (aggregation phase), δεν απαιτούνται atomic types ή locks.

```
// Use atomic types or explicit locks if accessed outside the aggregation phase
// For this pattern (aggregate after join), regular types are okay for globals
uint64_t total_reads_global = 0;
uint64_t total_writes_global = 0;
uint64_t total_read_time_us = 0; // Store time in microseconds
uint64_t total_write_time_us = 0; // Store time in microseconds
```

ΜΥΥ601 Λειτουργικά Συστήματα

Οι βιβλιοθήκες `pthread.h`, `time.h`, `inttypes.h` και `sys/time.h` εισάγονται για την υποστήριξη πολυνηματισμού, ακριβούς μέτρησης χρόνου και ασφαλούς μορφοποίησης 64-bit τιμών. Συγκεκριμένα, χρησιμοποιούνται για δημιουργία threads, υπολογισμό χρόνου λειτουργιών (σε microseconds) και εκτύπωση μεγάλων αριθμών με `PRIu64`.

```
#include <pthread.h>
#include <time.h>      // For clock_gettime
#include <inttypes.h>   // For PRIu64, PRId64 format specifiers
#include <sys/time.h>  // Keep for gettimeofday if needed elsewhere, or for seeding rand
```

Αρχείο Bench.h

Το αρχείο `bench.h` παρέχει τις ορισμούς δομών και πρωτότυπων συναρτήσεων που χρησιμοποιούνται σε μια πολυνηματική δοκιμαστική εφαρμογή (benchmark) για την μηχανή αποθήκευσης δεδομένων. Ο σκοπός της εφαρμογής είναι να δοκιμάσει και να αξιολογήσει την απόδοση της μηχανής υπό διάφορες συνθήκες και φόρτους εργασίας, επιτρέποντας την εκτέλεση αναγνώσεων, εγγραφών, και μικτών λειτουργιών. Πιο συγκεκριμένα, στο `bench.h` περιλαμβάνονται ορισμοί δομών όπως `Arguments`, `ThreadArg`, `TimeTracker`, και `Metrics`, οι οποίες καθορίζουν τις παραμέτρους της δοκιμής, τις ατομικές εντολές που εκτελούνται από κάθε νήμα, τη χρονομέτρηση των λειτουργιών, και τις συνολικές μετρήσεις απόδοσης αντίστοιχα.

Αρχείο 'db.c':

Αλλαγές στο αρχείο db.c

Οι αλλαγές στο `db.c` επικεντρώνονται στην προσθήκη συγχρονισμού μεταξύ πολλών νημάτων κατά την εκτέλεση των βασικών λειτουργιών της βάσης. Χρησιμοποιήθηκε η τεχνική των `read-write locks` (`pthread_rwlock_t`) ώστε να μπορούν να πραγματοποιούνται πολλές ταυτόχρονες αναγνώσεις χωρίς να επηρεάζονται μεταξύ τους, ενώ οι εγγραφές να γίνονται με ασφάλεια και αποκλειστικότητα. Συγκεκριμένα, προστέθηκε αποκλειστικό κλείδωμα στην `db_add` για τις εγγραφές και κοινό κλείδωμα στην `db_get` για τις αναγνώσεις. Η χρήση των `pthread_rwlock` επιτρέπει καλύτερη απόδοση σε περιβάλλοντα με μεγάλο αριθμό αναγνωστικών λειτουργιών, καθώς δεν απαιτεί πλήρες μπλοκάρισμα για κάθε πρόσβαση. Επίσης, έγινε σωστή αρχικοποίηση και καταστροφή του `lock` στη `db_open_ex` και `db_close` αντίστοιχα, διασφαλίζοντας έτσι την ορθή διαχείριση πόρων.

1. Προσθήκη μηχανισμού συγχρονισμού με pthread_rwlock

Προστέθηκε χρήση αναγνώστη/εγγραφέα κλειδώματος (`pthread_rwlock_t`) στη δομή DB και στις συναρτήσεις `db_add` και `db_get`.

ΜΥΥ601 Λειτουργικά Συστήματα

Αυτός ο μηχανισμός επιτρέπει την ασφαλή πρόσβαση σε κοινές δομές δεδομένων από πολλαπλά νήματα, χωρίς τη χρήση απλών mutex.

Έτσι, πολλαπλοί αναγνώστες μπορούν να διαβάζουν ταυτόχρονα, ενώ οι εγγραφές γίνονται σειριακά. Αυτό ενισχύει την ταυτόχρονη απόδοση χωρίς να διακινδυνεύεται η ακεραιότητα των δεδομένων.

```
#include <assert.h>
#include "db.h"
#include <pthread.h> // gia ta rwlocks, prepei na forwtthei h vivliothiki gia na doulepsi to sygchronisma
```

- **#include <pthread.h> // gia ta rwlocks:** Προσθέτει την απαραίτητη βιβλιοθήκη για να χρησιμοποιήσουμε pthread_rwlock_t.

→ *Βρίσκεται στη: Εκτός συναρτήσεων (στο πάνω μέρος του αρχείου)*

```
pthread_rwlock_init(&self->rwlock, NULL); // arxikopoioyme to read-write lock sto anoigma ths vashs
```

- **pthread_rwlock_init(&self->rwlock, NULL);:** Αρχικοποιεί το lock στο άνοιγμα της βάσης.

→ *Βρίσκεται μέσα στη συνάρτηση: db_open_ex*

```
pthread_rwlock_destroy(&self->rwlock); // katastrefoume to lock sto kleisimo gia na mh meinei memory leak
```

- **pthread_rwlock_destroy(&self->rwlock);:** Καταστρέφει το lock κατά το κλείσιμο της βάσης.

→ *Βρίσκεται στη: Μέσα στη συνάρτηση: db_close*

2. Κλείδωμα εγγραφής στη db_add

Η κλήση pthread_rwlock_wrlock προστέθηκε στην αρχή της db_add για να διασφαλιστεί ότι μόνο ένα νήμα μπορεί να γράψει ή να κάνει compact ταυτόχρονα.

Έπειτα από την εγγραφή, γίνεται ξεκλείδωμα με pthread_rwlock_unlock. Αυτή η αλλαγή είναι απαραίτητη για να αποφευχθούν race conditions στη memtable.

```
pthread_rwlock_wrlock(&self->rwlock); // kleidwnoume gia egrapfh (exclusive access) oste na mhn yparxoun alla thread pou grafoun h diavazoun
```

- **pthread_rwlock_wrlock(&self->rwlock);:** Αποκλειστικό κλείδωμα για εγγραφή στη βάση.

→ *Βρίσκεται στη: Μέσα στη συνάρτηση: db_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);

    skipplist_acquire(self->memtable->list); // auxanoume ton refcount ths skipplist gia na mhn kanoume free prin na prepei
    sst_merge(self->sst, self->memtable);
    self->memtable->compacted = 1; // simadeyoume oti h memtable exei kanei compact, xrhsimo gia debugging h logikh systimatos
    memtable_reset(self->memtable);
}
```

- **skipplist_acquire(self->memtable->list);**(προστέθηκε εντός db_add) Αυξάνει το refcount της skipplist, ώστε να μην αποδεσμευτεί πρόωρα κατά τη συγχώνευση.

•

- **self->memtable->compacted = 1;**Δηλώνει ότι η memtable έχει συγχωνευθεί. Μπορεί να χρησιμοποιηθεί για έλεγχο της κατάστασης αργότερα ή για logging.

```
pthread_rwlock_unlock(&self->rwlock); // telwnoume tin egragh kai kanoume unlock gia na proxorhsoun ta ypoloipa threads
return result;
```

- **pthread_rwlock_unlock(&self->rwlock);:** Ξεκλειδώνει μετά την ολοκλήρωση της εγγραφής.
→ Βρίσκεται στη: Μέσα στη συνάρτηση: db_add

3. Κλείδωμα ανάγνωσης στη db_get

Προστέθηκε η κλήση pthread_rwlock_rdlock στη db_get για να επιτραπεί ταυτόχρονη ανάγνωση από πολλαπλά νήματα με ασφαλή τρόπο.

Εφόσον δεν γίνονται αλλαγές στη memtable κατά την ανάγνωση, αυτή η χρήση είναι πιο αποδοτική από αποκλειστικά locks.

```
int db_get(DB* self, Variant* key, Variant* value)
{
    pthread_rwlock_rdlock(&self->rwlock); // kleidwma gia anagnwsh, epitrepei polla parallha reads alla oxi writes

    int found = 0;
    if (memtable_get(self->memtable->list, key, value) == 1)
        found = 1;
    else
        found = sst_get(self->sst, key, value);

    pthread_rwlock_unlock(&self->rwlock); // telos anagnwshs
    return found;
}
```

- **pthread_rwlock_rdlock(&self->rwlock);:** Κλείδωμα ανάγνωσης στη συνάρτηση db_get.
→ Βρίσκεται στη: Μέσα στη συνάρτηση: db_get
- **pthread_rwlock_unlock(&self->rwlock);:** Τέλος ανάγνωσης

4. Αρχικοποίηση και καταστροφή του lock

Η μεταβλητή pthread_rwlock προστέθηκε στη δομή DB και αρχικοποιείται στη db_open_ex με pthread_rwlock_init.

Αντίστοιχα, καταστρέφεται στη db_close με pthread_rwlock_destroy, ώστε να διασφαλίζεται ο καθαρισμός των πόρων.

- **pthread_rwlock_init(&self->rwlock, NULL);:** Αρχικοποιεί το lock στο άνοιγμα της βάσης.
→ Βρίσκεται μέσα στη συνάρτηση: db_open_ex
- **pthread_rwlock_destroy(&self->rwlock);:** Καταστρέφει το lock κατά το κλείσιμο της βάσης.
→ Βρίσκεται στη: Μέσα στη συνάρτηση: db_close
- **pthread_rwlock_t rwlock; :** Αυτό δηλώνει το lock για όλο το object της βάσης δεδομένων.Βρίσκεται στην δομή struct DB.

Αρχείο 'db.h':

Αλλαγές στο αρχείο db.h

Οι αλλαγές που έγιναν στο αρχείο db.h είχαν βασικό στόχο την υποστήριξη πολυνηματικής εκτέλεσης με ασφάλεια. Προστέθηκε η μεταβλητή pthread_rwlock_t rwlock στη δομή DB για να μπορεί να χρησιμοποιηθεί από τις συναρτήσεις της βάσης και να ρυθμίζει την ταυτόχρονη πρόσβαση. Επιπλέον, έγινε προσθήκη του include pthread.h ώστε να μπορεί να χρησιμοποιηθεί ο παραπάνω τύπος. Χωρίς αυτές τις δύο προσθήκες, οι αλλαγές που έγιναν στο db.c δεν θα μπορούσαν να λειτουργήσουν, καθώς στηρίζονται στον μηχανισμό συγχρονισμού μέσω read-write locks. Με αυτές τις προσθήκες, η μηχανή αποθήκευσης είναι πλέον πιο κατάλληλη για περιβάλλοντα με πολλαπλά νήματα και αυξημένο φόρτο εργασίας.

1. Προσθήκη pthread_rwlock στο struct DB

Προστέθηκε η μεταβλητή pthread_rwlock_t rwlock μέσα στη δομή DB για την υποστήριξη πολυνηματικής λειτουργίας.

Η μεταβλητή αυτή χρησιμοποιείται στη db.c για να διασφαλίζει ότι πολλαπλά νήματα μπορούν να προσπελάσουν με ασφάλεια τη βάση, επιτρέποντας ταυτόχρονες αναγνώσεις και αποκλειστικές εγγραφές.

- **pthread_rwlock_t rwlock; // gia lock metaksy add/get na mhn tsakwnontai:** Ορίζει ένα lock τύπου pthread_rwlock μέσα στη δομή DB για ασφαλή πολυνηματική πρόσβαση στις συναρτήσεις add/get.

2. Προσθήκη include pthread.h

Προστέθηκε η εντολή #include <pthread.h> στην αρχή του αρχείου για να είναι διαθέσιμος ο τύπος δεδομένων pthread_rwlock_t και οι σχετικές συναρτήσεις χειρισμού κλειδώματος.

- **pthread_rwlock_t rwlock; // gia lock metaksy add/get na mhn tsakwnontai:** Ορίζει ένα lock τύπου pthread_rwlock μέσα στη δομή DB για ασφαλή πολυνηματική πρόσβαση στις συναρτήσεις add/get.

Αναλυτική Περιγραφή Αλλαγών στα skipList.c, memtable.c, arena.c

Εισαγωγή

Στο πλαίσιο της επέκτασης και βελτιστοποίησης της λειτουργίας της μηχανής αποθήκευσης, πραγματοποιήθηκαν σημαντικές τροποποιήσεις στα αρχεία skipList.c, memtable.c, log.c και arena.c. Οι αλλαγές επικεντρώνονται στη σωστή διαχείριση της μνήμης, στην ασφαλή πολυνηματική προσπέλαση και στην αποφυγή διπλών αποδεσμεύσεων που οδηγούν σε αστάθεια. Επιπλέον, ενισχύθηκε η διαφάνεια και η ευκολία εντοπισμού σφαλμάτων μέσω κατάλληλων προειδοποιήσεων και σχολίων στον κώδικα. Παρακάτω καταγράφονται αναλυτικά οι γραμμές που που προστέθηκαν ή τροποποιήθηκαν:

skipList.c

Οι αλλαγές στο skipList.c επικεντρώνονται στην ασφαλή διαχείριση μνήμης και στην υποστήριξη πολυνηματικής λειτουργίας μέσω reference counting.

Η προσθήκη των συναρτήσεων skipList_acquire και skipList_release επιτρέπει την ασφαλή κοινή χρήση μιας skipList από πολλά components χωρίς να υπάρχει κίνδυνος να αποδεσμευτεί πρόωρα. Επιπλέον, η προστασία μέσω pthread_mutex διασφαλίζει ότι η προσπέλαση του refcount είναι thread-safe. Οι έλεγχοι για διπλή εγγραφή ή λανθασμένη αποδέσμευση προσθέτουν ασφάλεια και σταθερότητα στη δομή. Η χρήση της skipnode_size βοηθά στη μέτρηση του χώρου που καταλαμβάνουν οι κόμβοι, γεγονός που μπορεί να χρησιμοποιηθεί για debugging ή profiling.

- **void skipList_acquire(SkipList* self)**
→ Ορίζει νέα συνάρτηση για αύξηση του refcount με χρήση mutex για thread-safety.
- **void skipList_release(SkipList* self)**
→ Ορίζει νέα συνάρτηση για αποδέσμευση skipList με προστασία refcount και log σε περίπτωση κακής χρήσης.
- **pthread_mutex_lock(&self->lock);**
→ Προστατεύει τον refcount από ταυτόχρονες προσβάσεις.
- **fprintf(stderr, "[WARNING] Tried to release skipList with refcount=%d\n", self->refcount);**
→ Προειδοποιεί για πιθανό λανθασμένο διπλό release.
- **// Τα δεδομένα έχουν δεσμευτεί με arena_alloc -> δεν κάνουμε free στο data**
→ Επεξηγεί γιατί δεν αποδεσμεύουμε χειροκίνητα δεδομένα που διαχειρίζεται η arena.
- **static size_t skipnode_size(SkipNode* node)**
→ Νέα συνάρτηση για να υπολογίζει το μέγεθος των δεδομένων ενός κόμβου.
- **if (x != self->hdr && cmp_eq(x->data, key, klen)) return STATUS_OK;**
→ Αγνοεί διπλές εγγραφές ίδιου κλειδιού αντί να αντικαθιστά.

memtable.c

Οι αλλαγές στο memtable.c εξυπηρετούν τη διαχείριση κύκλου ζωής της skipool, η οποία αποτελεί την κύρια δομή αποθήκευσης του memtable.

Με την προσθήκη skipool_acquire κατά τη δημιουργία και skipool_release κατά την καταστροφή ή επανεκκίνηση της memtable, αποφεύγεται η πρόωρη διαγραφή της skipool. Επιπλέον, αφαιρέθηκε η αποδέσμευση μνήμης (free) μετά από insert, αφού πλέον η μνήμη παραμένει εντός της skipool μέσω της χρήσης της arena. Αυτό αποτρέπει διπλές αποδέσμευσεις και αυξάνει τη σταθερότητα της βάσης.

- **skipool_acquire(self->list);**
→ Καλείται στη δημιουργία/reset memtable για να αυξήσει το refcount της skipool.
- **skipool_release(self->list);**
→ Καλείται πριν αντικαταστήσουμε ή διαγράψουμε την skipool για σωστή διαχείριση μνήμης.
- **/*if (skipool_insert(...) == STATUS_OK_DEALLOC) free(mem);*/**
→ Σχολιάστηκε για αποφυγή διπλής αποδέσμευσης αφού τα δεδομένα μένουν στη skipool.

arena.c

Οι αλλαγές στο arena.c σχετίζονται με τον πιο ασφαλή και ελεγχόμενο τρόπο σαποδέσμευσης και ανακατανομής μνήμης.

Το σχόλιο στην pool_free υπενθυμίζει την αλλαγή που έγινε στο πώς απελευθερώνονται οι κόμβοι,

ενώ στην arena_realloc διορθώθηκε η λογική επανατοποθέτησης (reallocation) ώστε να επιστρέψει σωστά τον νέο pointer.

Αυτές οι αλλαγές συμβάλλουν στην αποφυγή σφαλμάτων μνήμης, όπως invalid free ή χρήση απελευθερωμένης μνήμης (use-after-free).

- **return newptr; // αυτό είναι το σωστό**
→ Διορθώνει τη λογική επιστροφής νέου pointer στην realloc όταν δεν υπάρχει αρκετή μνήμη.

Makefile.c

Η Makefile καθορίζει πώς θα μεταγλωττιστούν τα αρχεία του project και πώς θα παραχθεί η στατική βιβλιοθήκη libindexer.a. Μετά τις από τις προσθήκες μας αυτοματοποιεί τη διαδικασία build, ενσωματώνει πολυνηματική υποστήριξη μέσω -pthread (το οποίο προσθέσαμε), και προσφέρει εντολές για compilation, dependency generation και cleanup. Χρησιμοποιείται σε συνδυασμό με το defs.mk για κοινές παραμέτρους μεταγλώττισης.

```
FINAL_CFLAGS = $(STD) $(WARN) $(OPT) $(DEBUG) $(CFLAGS) $(DISWARN) -pthread  
WARN=-Wall -Wno-implicit-function-declaration -Wno-unused-but-set-variable  
DEBUG=-g -ggdb  
  
all:  
    gcc -g -ggdb -Wall -I../engine -Wno-implicit-function-declaration -Wno-unused-but-  
set-variable bench.c kiwi.c -L ../engine -lindexer -lpthread -lsnappy -o kiwi-bench  
clean:  
    rm -f kiwi-bench  
    rm -rf testdb
```

B. Που αποσκοπούν αυτές οι αλλαγές

Είσαγωγή

Σε αυτή την ενότητα της αναφοράς γίνεται μια συνοπτική αλλά ουσιαστική παρουσίαση των βασικών τεχνικών βελτιώσεων που εφαρμόστηκαν στον κώδικα. Το κύριο σκεπτικό πίσω από τις αλλαγές ήταν η ανάγκη για υποστήριξη λειτουργιών υπό συνθήκες φόρτου, όπως αυτές προκύπτουν σε πραγματικά περιβάλλοντα όπου πολλές διεργασίες αλληλεπιδρούν με τη βάση ταυτόχρονα.

Οι δύο βασικοί άξονες στους οποίους εστιάσαμε είναι η **πολυνηματική λειτουργία** και ο **συγχρονισμός**. Μέσα από την υλοποίηση αυτών των μηχανισμών, διασφαλίζεται τόσο η αποδοτικότητα όσο και η ασφάλεια της μηχανής αποθήκευσης κατά την ταυτόχρονη εκτέλεση πολλών reads και writes. Στις παραγράφους που ακολουθούν αναλύονται τα βήματα που ακολουθήθηκαν και τα τεχνικά εργαλεία που χρησιμοποιήθηκαν.

Πολυνηματική Λειτουργία

Για την υποστήριξη πολυνηματικής λειτουργίας, προστέθηκε μηχανισμός δημιουργίας νημάτων με χρήση της pthread_create, επιτρέποντας την ταυτόχρονη εκτέλεση πολλών λειτουργιών ανάγνωσης και εγγραφής από διαφορετικά νήματα. Η λειτουργία αυτή ενεργοποιείται μέσω της επιλογής mix κατά την εκκίνηση του προγράμματος, στην οποία ο χρήστης καθορίζει τον αριθμό των νημάτων, το ποσοστό εγγραφών (write ratio) και αν θα χρησιμοποιηθούν τυχαία ή προκαθορισμένα κλειδιά. Για κάθε νήμα, δημιουργείται μια ανεξάρτητη ροή εκτέλεσης που αναλαμβάνει να πραγματοποιήσει συγκεκριμένο αριθμό από λειτουργίες, με ξεχωριστή μέτρηση του χρόνου για reads και writes.

Στο τέλος, τα αποτελέσματα συγχωνεύονται ώστε να παραχθεί συνολική εικόνα της απόδοσης του συστήματος, μετρώντας συνολικό χρόνο, αριθμό reads/writes, και συνολικό throughput. Αυτή η δομή μάς δίνει τη δυνατότητα να προσομοιώσουμε

ρεαλιστικά σενάρια χρήσης όπου η μηχανή αποθήκευσης λειτουργεί υπό παράλληλο φόρτο, όπως θα συνέβαινε σε ένα πραγματικό περιβάλλον server.

Συγχρονισμός (Synchronization)

Για την αποφυγή προβλημάτων ταυτόχρονης πρόσβασης στα δεδομένα από πολλαπλά νήματα, ενσωματώθηκε μηχανισμός συγχρονισμού με χρήση pthread_rwlock_t. Συγκεκριμένα, η συνάρτηση db_get που αφορά αναγνώσεις, χρησιμοποιεί pthread_rwlock_rdlock, το οποίο επιτρέπει πολλαπλά reads να συμβαίνουν ταυτόχρονα. Αντίθετα, η db_add που αφορά εγγραφές, χρησιμοποιεί pthread_rwlock_wrlock, επιτρέποντας αποκλειστική πρόσβαση σε ένα μόνο νήμα κάθε φορά. Έτσι αποφεύγονται καταστάσεις race conditions ή ασυνέπειας δεδομένων.

Η διαχείριση του κύκλου ζωής του lock γίνεται σωστά μέσω των pthread_rwlock_init και pthread_rwlock_destroy, οι οποίες καλούνται στις db_open και db_close αντίστοιχα. Αυτή η προσέγγιση προσφέρει έναν αξιόπιστο και αποδοτικό τρόπο συγχρονισμού, ειδικά σε εφαρμογές όπου οι αναγνώσεις είναι πολύ συχνότερες από τις εγγραφές

Γ. Logs και έλεγχος ορθότητας

Στην παρούσα ενότητα παραθέτουμε στοιχεία που συλλέχθηκαν κατά την εκτέλεση των δοκιμών και αφορούν την **ορθή λειτουργία και σταθερότητα της μηχανής αποθήκευσης**. Μέσω αρχείων καταγραφής (.log) και επιλεγμένων screenshots, επιβεβαιώσαμε ότι οι κρίσιμες λειτουργίες του συστήματος —όπως η διαχείριση της skip list, η δημιουργία SST αρχείων και η διαδικασία compaction— εκτελούνται ομαλά και σύμφωνα με το σχεδιασμό. Ο έλεγχος ορθότητας βασίστηκε σε ενδείξεις σωστού synchronization, ορθής διαχείρισης μνήμης και απουσίας σφαλμάτων κατά τη διάρκεια εκτέλεσης, εξασφαλίζοντας ότι η υλοποίηση είναι αξιόπιστη ακόμα και σε σενάρια πολυνηματικής πίεσης.

Αποτελέσματα(screenshots):

Στις παρακάτω φωτογραφίες παρουσιάζονται τα αποτελέσματα από την εκτέλεση του πολυνηματικού benchmarking για διαφορετικούς συνδυασμούς παραμέτρων. Πιο συγκεκριμένα, κάθε πίνακας δείχνει για ένα συγκεκριμένο πλήθος εγγραφών (Entries) την επίδοση της μηχανής αποθήκευσης ως προς τον αριθμό νημάτων (Threads), το ποσοστό εγγραφών/αναγνώσεων (W/R %), τον ρυθμό επεξεργασίας (Throughput σε ops/s), τη μέση καθυστέρηση (Average Latency), καθώς και ξεχωριστά το latency για αναγνώσεις και εγγραφές. Επιπλέον, περιλαμβάνεται και η συνολική χρονική διάρκεια του κάθε

ΜΥΥ601 Λειτουργικά Συστήματα

πειράματος. Τα δεδομένα αυτά επιτρέπουν την αξιολόγηση της συμπεριφοράς του συστήματος υπό διαφορετικά επίπεδα φόρτου και παραλληλίας.

ΠΑΡΑΤΗΡΗΣΗ: Για να δείτε καλύτερα τους παρακάτω πίνακες πολύ πιθανόν να χρειαστεί zoom!

Entries	W/R (%)	Threads	Thrput (ops/s)	Avg Lat (μs)	Read Lat (μs)	Write Lat (μs)	Time (s)
100.000	20/80	3	111203	21.1	10.1	65.4	0.899
100.000	20/80	5	191626	20.1	5.7	78.0	0.522
100.000	20/80	10	157100	54.1	7.0	242.5	0.657
100.000	20/80	20	235012	67.7	4.7	320.3	0.426
100.000	20/80	25	161808	110.1	6.5	525.0	0.618
100.000	50/50	3	140971	18.6	8.5	28.9	0.709
100.000	50/50	5	217829	17.3	4.9	29.8	0.459
100.000	50/50	10	151938	52.7	8.3	97.3	0.658
100.000	50/50	20	230632	69.9	5.3	134.5	0.434
100.000	50/50	25	210169	99.7	5.6	181.7	0.476
100.000	80/20	3	201277	13.3	7.1	14.8	0.497
100.000	80/20	5	200950	20.6	6.0	24.3	0.498
100.000	80/20	10	186439	43.2	7.7	52.0	0.536
100.000	80/20	20	194584	79.4	12.4	96.3	0.514
100.000	80/20	25	153942	128.1	14.4	156.6	0.650

Entries	W/R (%)	Threads	Thrput (ops/s)	Avg Lat (μs)	Read Lat (μs)	Write Lat (μs)	Time (s)
300.000	20/80	3	127950	19.1	8.6	61.0	2.345
300.000	20/80	5	167777	23.5	6.3	91.9	1.788
300.000	20/80	10	162763	52.1	7.0	230.7	1.943
300.000	20/80	20	178788	100.0	6.1	473.8	1.678
300.000	20/80	25	183289	118.2	6.3	565.2	1.637
300.000	50/50	3	171893	15.8	6.1	25.6	1.745
300.000	50/50	5	187025	24.5	5.5	43.6	1.604
300.000	50/50	10	182300	46.8	6.1	87.7	1.646
300.000	50/50	20	165634	94.3	7.7	180.3	1.811
300.000	50/50	25	180073	115.7	7.8	223.1	1.666
300.000	80/20	3	154692	18.5	14.4	19.6	1.939
300.000	80/20	5	170944	27.3	10.0	31.6	1.755
300.000	80/20	10	192772	45.6	9.8	54.6	1.557
300.000	80/20	20	176984	95.7	9.8	117.2	1.701

Entries	W/R (%)	Threads	Thrput (ops/s)	Avg Lat (μs)	Read Lat (μs)	Write Lat (μs)	Time (s)
600.000	20/80	3	163919	17.5	6.4	61.7	3.660
600.000	20/80	5	167244	28.8	6.5	117.9	3.588
600.000	20/80	10	143752	61.6	8.0	275.9	4.174
600.000	20/80	20	148943	119.6	7.8	567.5	4.028
600.000	20/80	25	158629	141.8	7.5	678.7	3.792
600.000	50/50	3	161713	17.7	7.0	28.3	3.710
600.000	50/50	5	121926	37.0	10.2	63.8	4.765
600.000	50/50	10	161552	57.7	8.3	107.1	3.714
600.000	50/50	20	160654	110.1	8.1	212.3	3.735
600.000	50/50	25	145407	154.2	9.6	299.1	4.126
600.000	80/20	3	141638	19.4	12.0	21.3	4.177
600.000	80/20	5	161995	28.8	11.9	33.0	3.704
600.000	80/20	10	154093	60.9	12.7	73.0	3.894
600.000	80/20	20	146966	127.2	13.5	155.6	4.003
600.000	80/20	25	148560	152.7	15.4	187.1	4.039

ΜΥΥ601 Λειτουργικά Συστήματα

Entries	W/R (%)	Threads	Thruput (ops/s)	Avg Lat (μs)	Read Lat (μs)	Write Lat (μs)	Time (s)
1.000.000	20/80	3	113208	23.3	9.8	77.4	8.833
1.000.000	20/80	5	125945	37.5	8.9	151.9	7.946
1.000.000	20/80	10	133652	69.3	8.6	312.6	7.402
1.000.000	20/80	20	132440	139.2	8.7	660.4	7.551
1.000.000	20/80	25	127154	181.5	9.3	870.7	7.864
1.000.000	50/50	3	151020	19.0	7.7	30.3	6.622
1.000.000	50/50	5	132690	35.6	9.6	61.5	7.536
1.000.000	50/50	10	151790	62.9	8.2	117.6	6.598
1.000.000	50/50	20	155594	121.1	8.9	233.2	6.427
1.000.000	50/50	25	155517	148.9	8.2	289.2	6.430
1.000.000	80/20	3	145486	19.6	12.3	21.4	6.873
1.000.000	80/20	5	150414	31.3	12.4	36.0	6.648
1.000.000	80/20	10	160021	60.0	14.1	71.5	6.218
1.000.000	80/20	20	143030	132.0	15.7	161.2	6.953
1.000.000	80/20	25	136108	169.2	15.5	207.7	7.336

Συμπεράσματα:

Κατά την εκτέλεση των πειραμάτων με διαφορετικά ποσοστά ανάγνωσης/εγγραφής, αριθμό νημάτων και πλήθος δεδομένων, παρατηρήθηκε πως η μηχανή αποθήκευσης παρουσιάζει ικανοποιητική απόδοση υπό ρεαλιστικές συνθήκες φόρτου. Όταν οι αναγνώσεις υπερτερούν των εγγραφών (π.χ. 20/80 W/R), το σύστημα επιτυγχάνει υψηλό throughput και χαμηλό latency, γεγονός που υποδηλώνει αποδοτική αξιοποίηση των πόρων σε read-heavy σενάρια. Αντίθετα, σε write-heavy περιπτώσεις (80/20 W/R), ο χρόνος απόκρισης των εγγραφών αυξάνεται σημαντικά, ενώ το συνολικό throughput μειώνεται, κάτι που αποδίδεται στο synchronization overhead και τις αυξημένες απαιτήσεις διαχείρισης μνήμης.

Επιπλέον, η αύξηση του αριθμού νημάτων οδηγεί σε βελτίωση της απόδοσης μέχρι ένα σημείο, πέρα από το οποίο εμφανίζονται φαινόμενα κορεσμού, όπως σταθεροποίηση ή πτώση του throughput και αύξηση της καθυστέρησης. Αυτό αποδεικνύει ότι η εφαρμογή είναι σχεδιασμένη να κλιμακώνεται σωστά μέχρι έναν βαθμό, ωστόσο επηρεάζεται από το contention στους shared πόρους. Η συνολική συμπεριφορά της μηχανής επιβεβαιώνει τη σωστή υλοποίηση των μηχανισμών πολυνηματικής υποστήριξης και συγχρονισμού, προσφέροντας αξιόπιστα και σταθερά αποτελέσματα ακόμα και σε συνθήκες μεγάλης ταυτόχρονης πρόσβασης.