

Υλοποίηση πολυνηματικής λειτουργίας σε μηχανή αποθήκευσης δεδομένων

ΜΥΥ601 Λειτουργικά Συστήματα

Σκαραφίγκας Βασίλειος

Περιεχόμενα

1. Εισαγωγή	3
1.1 Γενικά	3
1.1.1 Στόχος Εργασίας.....	3
2. Παράμετροι.....	4
2.1 Βασικές Δομές Παραμέτρων	4
2.1.1 Δομή BasicBenchArgs	4
2.1.2 Δομή AdvancedBenchArgs	4
2.1.3 Δομή DefaultArgs	5
2.2 Διαχείριση Σφαλμάτων (Error Handling).....	5
2.2.1 Δομή ErrorHandler	5
2.2.2 Δομή ErrorCode.....	5
2.3 Ανάλυση και Επεξήγηση των Παραμέτρων.....	6
2.3.1 Συνάρτηση Εξέτασεις Παραμέτρων (parseBenchArgs)	6
2.3.2 Πίνακας Παραδειγμάτων	8
2.3.3 Στιγμιότυπα Σφαλμάτων	9
2.3.4 Ροή Προγράμματος main().....	10
3. Πολυνηματική Προετοιμασία.....	11
3.1 Ανάλυση Βασικών Δομών	11
3.1.1 Δομή ThreadBenchArgs	11
3.1.2 Δομή BenchResults	12
3.1.3 Δομή Datas.....	12
3.2 Κατανομή των εργασιών μεταξύ των threads (partitionWorkload)	13
3.3 Χρονομέτρηση – Κόστος	15
3.3.1 Δομή TimerData.....	15
3.3.2 Τύποι χρόνου μέτρησης	15
3.3.3 Ακρίβεια στη μέτρησης.....	16
3.3.4 Ειδική περίπτωση μέτρησης.....	16

3.3.5 Πώς χειρίζομαστε τις ακραίες περιπτώσεις.....	17
3.4 Ροή Εκτέλεσης Διεργασιών runBenchmark.....	17
3.5 Αποτελέσματα.....	20
3.5.1 Υπολογισμός Αποτελεσμάτων (computeResults ()).....	20
3.5.2 Εκτύπωση Αποτελεσμάτων (printResults())	21
4 Σύστημα και Τροποποιήσεις.....	23
4.1 Προδιαγραφές Συστήματος	23
4.2 Αρχεία Κλειδαριών και Makefile	23
4.3 Compile.....	24
5. Αμοιβαίος Αποκλεισμός.....	24
5.1 Γενικά	24
5.1.1 Αναγκαιότητα του αμοιβαίου αποκλεισμού	25
5.1.2 Τύποι κλειδώματος και στρατηγικές	25
5.2 Ο δικός μας μηχανισμός (RWLocker).....	25
5.2.2 Δομή Κλειδαριάς RWLocker.....	26
5.2.3 Ανάλυση αποκλεισμού και ταυτοχρονισμού	26
5.3 Ιδέες Βελτίωσης Μηχανισμού	30
6. Πειραματική Διαδικασία.....	31
6.1 Παράμετροι των Πειραμάτων και τρόπος μελέτης.....	31
6.2 Σχολιασμός Πειραμάτων Ανάγνωση	32
6.2.1 Συνολικός Χρόνος Εκτέλεσης (Read).....	32
6.2.2 Ρυθμαπόδοση (Throughput) - ops/sec (Read)	33
6.3 Σχολιασμός Πειραμάτων Εγγραφέων	34
6.3.1 Συνολικός Χρόνος Εκτέλεσης (Write)	34
6.3.2 Ρυθμαπόδοση (Throughput) - ops/sec (Write).....	35
6.4 Σχολιασμός Πειραμάτων Μεικτής Λειτουργίας (ReadWrite).....	36
6.4.1 Συνολικός Χρόνος (Total Time) για 70% Write / 30% Read	36
6.4.2 Ρυθμαπόδοση (ops/sec) για 70% Write / 30% Read	37
6.4.3 Συνολικός Χρόνος και Ρυθμαπόδοση (Total Time) για 50% Write / 50% Read..	37
6.4.4 Συνολικός Χρόνος και Ρυθμαπόδοση (Total Time) για 30% Write / 70% Read..	40

1. Εισαγωγή

1.1 Γενικά

Η αποθηκευτική μηχανή Kiwi αποτελεί μια σύγχρονη λύση διαχείρισης δεδομένων, σχεδιασμένη για να υποστηρίζει την αξιόπιστη και κλιμακούμενη αποθήκευση σε περιβάλλοντα νέφους, όπου απαιτείται υψηλή απόδοση και ταυτόχρονη επεξεργασία μεγάλου όγκου δεδομένων. Η βάση της μηχανής είναι το LSM-tree (Log-Structured Merge Tree), μια δομή δεδομένων που επιτρέπει την αποτελεσματική εισαγωγή και ανάκτηση ζευγών κλειδιού-τιμής. Στην υλοποίηση της Kiwi, η εισαγωγή των δεδομένων πραγματοποιείται αρχικά στο memtable, που συνήθως υλοποιείται ως skip list, εξασφαλίζοντας γρήγορη πρόσβαση στη μνήμη για τις πιο πρόσφατες καταχωρήσεις. Όταν το memtable γεμίσει, τα δεδομένα μεταφέρονται στον δίσκο και οργανώνονται σε Sorted String Tables (SST), όπου επιπλέον χρησιμοποιούνται bloom filters για την ταχύτερη απόρριψη μη σχετικών περιοχών κατά την αναζήτηση.

Παράλληλα, η μηχανή υλοποιεί μηχανισμούς συγχώνευσης (merging/compaction) των SST αρχείων, διαδικασία που πραγματοποιείται ασύγχρονα από ένα ξεχωριστό νήμα (merge thread). Αυτή η προσέγγιση συμβάλλει στη διατήρηση της συνοχής και της αποδοτικότητας των αποθηκευμένων δεδομένων, μειώνοντας την κατανομή των αρχείων σε πολλαπλά επίπεδα και εξασφαλίζοντας ότι οι πρόσβασεις στα δεδομένα παραμένουν ταχείες ακόμη και όταν το σύστημα βρίσκεται υπό φορτίο.

1.1.1 Στόχος Εργασίας

Ο κύριος στόχος της παρούσας εργασίας είναι να επεκτείνει την υπάρχουσα υλοποίηση της μηχανής Kiwi με την εφαρμογή ασφαλούς πολυνηματικής λειτουργίας στις εντολές **add** και **get**. Συγκεκριμένα, επιδιώκεται:

- Βελτίωση του Συγχρονισμού:** Να εξασφαλιστεί ότι πολλαπλά νήματα μπορούν να εκτελούν ταυτόχρονα τις λειτουργίες εισαγωγής και ανάκτησης χωρίς να δημιουργούνται καταστάσεις ανταγωνισμού (race conditions). Αυτό θα επιτευχθεί μέσω της χρήσης προχωρημένων τεχνικών συγχρονισμού, όπως ο αμοιβαίος αποκλεισμός (mutexes) και οι μεταβλητές συνθήκης (condition variables), που έχουν ήδη ενσωματωθεί σε διάφορα μέρη του κώδικα.
- Αξιολόγηση Απόδοσης:** Να συλλεχθούν και να αναλυθούν στατιστικά στοιχεία απόδοσης, τα οποία θα καταδείξουν τη διαφορά μεταξύ της αρχικής μονονηματικής προσέγγισης και της νέας πολυνηματικής υλοποίησης. Αυτό περιλαμβάνει τη μέτρηση χρόνων απόκρισης και την ανάλυση του throughput υπό διάφορα σενάρια φόρτου (π.χ., λειτουργίες μόνο add, μόνο get ή μίξη των δύο).

Η παρούσα εργασία, επομένως, επιδιώκει όχι μόνο να βελτιώσει την πολυνηματική λειτουργία του συστήματος, αλλά και να παρέχει μια πλήρη τεκμηρίωση των αλλαγών που πραγματοποιήθηκαν και των πειραματικών αποτελεσμάτων που επαληθεύουν την ορθότητα και την αποδοτικότητα της νέας υλοποίησης. Μέσω αυτής της προσέγγισης, επιδιώκεται η δημιουργία ενός συστήματος που θα ανταποκρίνεται στις απαιτήσεις των σύγχρονων υποδομών αποθήκευσης, όπου η ταυτόχρονη επεξεργασία δεδομένων και η αποδοτική διαχείριση των πόρων είναι κρίσιμες για την επιτυχία.

2. Παράμετροι

Η διαμόρφωση και σωστή διαχείριση των παραμέτρων αποτελεί θεμελιώδες τμήμα κάθε αξιόπιστης προσομοίωσης. Στο συγκεκριμένο κεφάλαιο αναλύεται εκτενώς ο τρόπος με τον οποίο γίνεται η αρχικοποίηση και επεξεργασία των παραμέτρων που δίνονται ως είσοδοι στη συνάρτηση εκκίνησης της προσομοίωσης (`parseBenchArgs`). Η λειτουργία αυτή έχει στόχο τη σαφή διάκριση μεταξύ έγκυρων και μη έγκυρων περιπτώσεων ορισμάτων, ώστε να διασφαλιστεί η ορθότητα και αξιοπιστία της προσομοίωσης.

Η συνάρτηση `parseBenchArgs` λαμβάνει μια σειρά ορισμάτων από τη γραμμή εντολών και τα αναλύει, αντιστοιχίζοντάς τα στις δομές δεδομένων που χρησιμοποιεί το πρόγραμμα για να καθορίσει τη συμπεριφορά του. Μέσα από αυτή την ανάλυση, επιτυγχάνεται η ευελιξία στη χρήση διαφορετικών συνδυασμών παραμέτρων, ενώ ταυτόχρονα προστατεύεται το πρόγραμμα από την είσοδο μη έγκυρων ή ατελών δεδομένων.

Για τον σκοπό αυτό, χρησιμοποιούνται οι δομές `BasicBenchArgs`, `AdvancedBenchArgs`, και `DefaultArgs`, οι οποίες περιέχουν τόσο υποχρεωτικές όσο και προαιρετικές παραμέτρους. Οι δομές αυτές αποτελούν το θεμέλιο για την ορθή οργάνωση και τη διαχείριση των δεδομένων που θα χρησιμοποιηθούν κατά την εκτέλεση της προσομοίωσης.

2.1 Βασικές Δομές Παραμέτρων

2.1.1 Δομή BasicBenchArgs

```
/* Δομή για τα βασικά ορίσματα του benchmark (υποχρεωτικά) */
typedef struct {
    char *operationMode;      // Λειτουργία: "read", "write" ή "readwrite"
    long int operationCount; // Συνολικός αριθμός εργασιών
} BasicBenchArgs;
```

Η δομή αυτή περιέχει τις υποχρεωτικές παραμέτρους:

- `operationMode`: Η λειτουργία που θα εκτελεστεί (`read`, `write`, ή `readwrite`).
- `operationCount`: Ο συνολικός αριθμός των εργασιών προς εκτέλεση.

2.1.2 Δομή AdvancedBenchArgs

```
typedef struct {
    long int threadCount;      // Συνολικός αριθμός των threads
    double readPercentage;     // Ποσοστό read, π.χ. 50% (μόνο για readwrite mode)
    int isRandom;              // Τυχαία πρόσβαση (1) ή σειριακή (0)
} AdvancedBenchArgs;
```

Η δομή αυτή περιλαμβάνει παραμέτρους που καθορίζουν την πιο εξειδικευμένη συμπεριφορά της προσομοίωσης:

- `threadCount`: Ο αριθμός των νημάτων (`threads`) που θα χρησιμοποιηθούν.
- `readPercentage`: Το ποσοστό των αναγνωστικών εργασιών έναντι των εγγραφών (σημαντικό μόνο για τη λειτουργία `readwrite`).
- `isRandom`: Καθορίζει αν η πρόσβαση στα δεδομένα θα είναι τυχαία (1) ή σειριακή (0).

2.1.3 Δομή DefaultArgs

```
/* Δομή για προεπιλεγμένες τιμές ορισμάτων */
typedef struct {
    long int defaultThreadCount;
    double defaultReadPercentage;
    int defaultRandom;
} DefaultArgs;
```

Αυτή η δομή παρέχει προεπιλεγμένες τιμές:

- defaultThreadCount: Προεπιλεγμένος αριθμός νημάτων (1 νήμα).
- defaultReadPercentage: Προεπιλεγμένο ποσοστό αναγνώσεων (50% για τη λειτουργία readwrite).
- defaultRandom: Προεπιλεγμένη επιλογή σειριακής πρόσβασης (0).

2.2 Διαχείριση Σφαλμάτων (Error Handling)

2.2.1 Δομή ErrorHandler

```
/* Δομή για πληροφορίες σφαλμάτων */
typedef struct {
    ErrorCode code;
    char message[256];
} ErrorHandler;
```

Η διαχείριση των έγκυρων και μη έγκυρων περιπτώσεων γίνεται μέσω της ειδικής δομής ErrorHandler και της συνάρτησης handleError. Η δομή ErrorHandler περιέχει τον κωδικό του σφάλματος (ErrorCode) και ένα μήνυμα που περιγράφει το σφάλμα.

2.2.2 Δομή ErrorCode

```
/* Ορισμός κωδικών σφαλμάτων */
typedef enum {
    ERR_NONE = 0,
    ERR_INSUFFICIENT_ARGS,
    ERR_INVALID_OPERATION,
    ERR_INVALID_COUNT,
    ERR_INVALID_THREADS,
    ERR_INVALID_READ_PERCENTAGE
} ErrorCode;
```

Κωδικοί Σφαλμάτων (ErrorCode):

- ERR_INSUFFICIENT_ARGS: Ανεπαρκή ορίσματα εισόδου.
- ERR_INVALID_OPERATION: Μη έγκυρη λειτουργία (read, write, readwrite).
- ERR_INVALID_COUNT: Αρνητικός ή μηδενικός αριθμός εργασιών.
- ERR_INVALID_THREADS: Μη έγκυρος αριθμός νημάτων.
- ERR_INVALID_READ_PERCENTAGE: Μη έγκυρο ποσοστό αναγνώσεων.

```

/* Συνάρτηση για διαχείριση σφάλμάτων με χρήση switch-case */
void handleError(const ErrorHandler err) {
    fprintf(stderr, "Benchmark Error: ");
    switch(err.code) {
        case ERR_INSUFFICIENT_ARGS:
            fprintf(stderr, "Άνεπαρκή ορισμάτων. Χρήση: ./kiwi-bench <read | write | readwrite> <count> [<threads> <random> [<read_percentage>]]\n");
            break;
        case ERR_INVALID_OPERATION:
            fprintf(stderr, "Μη έγκυρη λειτουργία. Επιλέξτε 'read', 'write' ή 'readwrite'.\n");
            break;
        case ERR_INVALID_COUNT:
            fprintf(stderr, "Ο αριθμός εργασιών πρέπει να είναι μεγαλύτερος του 0.\n");
            break;
        case ERR_INVALID_THREADS:
            fprintf(stderr, "Ο αριθμός νημάτων πρέπει να είναι μεγαλύτερος του 0. Για λειτουργία readwrite απαιτούνται τουλάχιστον 2 νήματα.\n");
            break;
        case ERR_INVALID_READ_PERCENTAGE:
            fprintf(stderr, "Το ποσοστό αναγνώσεων πρέπει να είναι > 0 και μικρότερο του 100.\n");
            break;
        default:
            fprintf(stderr, "Αγνωστο σφάλμα.\n");
            break;
    }
    exit(EXIT_FAILURE);
}

```

Λειτουργία handleError: Η συνάρτηση αυτή δέχεται μια μεταβλητή τύπου ErrorHandler, αναγνωρίζει τον τύπο του σφάλματος (code) και εμφανίζει ένα σαφές και επεξηγηματικό μήνυμα στον χρήστη. Μετά την εμφάνιση του μηνύματος, το πρόγραμμα τερματίζεται άμεσα για να αποφευχθεί οποιαδήποτε περαιτέρω ανεπιθύμητη εκτέλεση.

2.3 Ανάλυση και Επεξήγηση των Παραμέτρων

```

/* Συνάρτηση για ανάλυση των ορισμάτων (argc, argv) και ανάθεση στις δομές BasicBenchArgs και AdvancedBenchArgs */
void parseBenchArgs(int argc, char **argv, BasicBenchArgs *basic, AdvancedBenchArgs *advanced) {

```

Η συνάρτηση parseBenchArgs που βρίσκεται στο αρχείο **«bench.c»** είναι υπεύθυνη για την ανάλυση και την επεξεργασία των ορισμάτων που παρέχονται από τη γραμμή εντολών κατά την εκκίνηση της προσομοίωσης. Κεντρικός σκοπός της συνάρτησης είναι να εξασφαλίσει την ορθή και ασφαλή ανάθεση τιμών στις βασικές δομές δεδομένων BasicBenchArgs και AdvancedBenchArgs, βάσει των οποίων θα πραγματοποιηθεί η εκτέλεση των λειτουργιών της προσομοίωσης.

2.3.1 Συνάρτηση Εξέτασεις Παραμέτρων (parseBenchArgs)

Αρχικά, η συνάρτηση ορίζει ορισμένες προεπιλεγμένες τιμές ([DefaultArgs](#)) που χρησιμοποιούνται όταν ο χρήστης δεν παρέχει συγκεκριμένες προαιρετικές παραμέτρους. Οι τιμές αυτές είναι:

```

ErrorHandler err;
/* Ορισμός προεπιλεγμένων τιμών */
DefaultArgs defaults = {1, 50.0, 0}; // 1 νήμα, 50% read (για readwrite), random = 0

```

- defaultThreadCount: 1 νήμα
- defaultReadPercentage: 50% αναγνώσεων για τη λειτουργία readwrite
- defaultRandom: 0 (σειριακή πρόσβαση δεδομένων)

```

/* Έλεγχος συνολικού αριθμού ορισμάτων (πρέπει να είναι από 3 έως 6) */
if (argc < 3 || argc > 6) {
    err.code = ERR_INSUFFICIENT_ARGS;
    strcpy(err.message, "Δείποντας ή υπάρχουν επιπλέον ορισμάτα.");
    handleError(err);
}

```

Στη συνέχεια, η συνάρτηση εκτελεί έναν αρχικό έλεγχο ώστε να διαπιστωθεί αν ο αριθμός των ορισμάτων (argc) βρίσκεται μέσα στα αποδεκτά όρια (από 3 έως 6). Αν ο αριθμός των ορισμάτων είναι ανεπαρκής ή υπερβολικός, η συνάρτηση χρησιμοποιεί την ειδική δομή ErrorHandler για να δημιουργήσει ένα κατάλληλο μήνυμα λάθους (ERR_INSUFFICIENT_ARGS) και να τερματίσει το πρόγραμμα.

```

/* Ανάθεση βασικών ορισμάτων */
basic->operationMode = argv[1];
if (strcmp(basic->operationMode, "read") != 0 &&
    strcmp(basic->operationMode, "write") != 0 &&
    strcmp(basic->operationMode, "readwrite") != 0) {
    err.code = ERR_INVALID_OPERATION;
    strcpy(err.message, "Οι λειτουργίες είναι μόνο 'read', 'write' ή 'readwrite'.");
    handleError(err);
}

```

Κατόπιν, η συνάρτηση αναλύει το πρώτο όρισμα (argv[1]), που αντιπροσωπεύει τον τύπο της λειτουργίας (operationMode) που θα εκτελεστεί. Οι έγκυρες τιμές είναι "read", "write" και "readwrite". Αν το όρισμα δεν αντιστοιχεί σε καμία από αυτές τις τιμές, παράγεται ένα σφάλμα τύπου ERR_INVALID_OPERATION.

```

basic->operationCount = atol(argv[2]);
if (basic->operationCount <= 0) {
    err.code = ERR_INVALID_COUNT;
    strcpy(err.message, "Ο αριθμός εργασιών πρέπει να είναι μεγαλύτερος του 0.");
    handleError(err);
}

```

Αμέσως μετά, αναλύεται το δεύτερο όρισμα (argv[2]), που καθορίζει τον συνολικό αριθμό εργασιών (operationCount). Η τιμή αυτή πρέπει να είναι θετικός αριθμός. Αν το όρισμα είναι μηδενικό ή αρνητικό, παράγεται το σφάλμα ERR_INVALID_COUNT.

Ανάλογα με τον τύπο της λειτουργίας που έχει επιλεγεί, η συνάρτηση συνεχίζει την επεξεργασία των προαιρετικών ορισμάτων:

- **Για τη λειτουργία "readwrite":**

```

/* Επεξεργασία προαιρετικών ορισμάτων ανάλογα με τη λειτουργία */
if (strcmp(basic->operationMode, "readwrite") == 0) {
    if (argc < 4) {
        err.code = ERR_INSUFFICIENT_ARGS;
        strcpy(err.message, "Για λειτουργία readwrite απαιτείται τουλάχιστον <threads>.");
        handleError(err);
    }

    advanced->threadCount = atol(argv[3]);
    if (advanced->threadCount < 2) {
        err.code = ERR_INVALID_THREADS;
        strcpy(err.message, "Για λειτουργία readwrite απαιτούνται τουλάχιστον 2 νήματα.");
        handleError(err);
    }

    // Αν έχουμε τουλάχιστον 5 args, τότε το 5ο είναι το random
    if (argc >= 5) {
        advanced->isRandom = atoi(argv[4]);
    } else {
        advanced->isRandom = defaults.defaultRandom;
    }

    // Αν έχουμε 6 args, τότε το 6ο είναι το readPercentage
    if (argc == 6) {
        advanced->readPercentage = atof(argv[5]);
        if (advanced->readPercentage <= 0 || advanced->readPercentage >= 100) {
            err.code = ERR_INVALID_READ_PERCENTAGE;
            strcpy(err.message, "Το ποσοστό αναγνώσεων πρέπει να είναι > 0 και < 100.");
            handleError(err);
        }
    } else {
        advanced->readPercentage = defaults.defaultReadPercentage;
    }
}

```

- Το τρίτο όρισμα (argv[3]) αντιστοιχεί στον αριθμό των νημάτων (threadCount), που πρέπει να είναι τουλάχιστον 2. Σε αντίθετη περίπτωση, δημιουργείται σφάλμα ERR_INVALID_THREADS.
- Το τέταρτο όρισμα (argv[4]) αντιστοιχεί στο είδος της πρόσβασης στα δεδομένα (isRandom): τυχαία (1) ή σειριακή (0). Αν δεν δοθεί τιμή, χρησιμοποιείται η προεπιλεγμένη (0).
- Το πέμπτο όρισμα (argv[5]) καθορίζει το ποσοστό των αναγνώσεων (readPercentage). Αν η τιμή αυτή δεν δοθεί, επιλέγεται η προεπιλεγμένη (50%). Σε περίπτωση που δοθεί ποσοστό αναγνώσεων εκτός του διαστήματος (0,100), παράγεται σφάλμα τύπου ERR_INVALID_READ_PERCENTAGE.

- Για τη λειτουργία "read":

```

else if (strcmp(basic->operationMode, "read") == 0) {
    /* Όταν read: αν υπάρχει 4ο όρισμα, τότε αυτό είναι ο αριθμός νημάτων, και αν υπάρχει 5ο, αυτό το isRandom */
    if (argc >= 4) {
        advanced->threadCount = atol(argv[3]);
    } else {
        advanced->threadCount = defaults.defaultThreadCount;
    }
    if (advanced->threadCount <= 0) {
        err.code = ERR_INVALID_THREADS;
        strcpy(err.message, "Ο αριθμός νημάτων πρέπει να είναι μεγαλύτερος του 0.");
        handleError(err);
    }
    if (argc >= 5) {
        advanced->isRandom = atoi(argv[4]);
    } else {
        advanced->isRandom = defaults.defaultRandom;
    }
    advanced->readPercentage = 100.0; // Όλες οι εργασίες είναι αναγνώσεις.
}

```

- Το τρίτο όρισμα (argv[3]) καθορίζει τον αριθμό των νημάτων, με προεπιλεγμένη τιμή το 1 αν δεν δοθεί.
- Το τέταρτο όρισμα (argv[4]) είναι το είδος της πρόσβασης (τυχαία ή σειριακή), με προεπιλεγμένη τιμή τη σειριακή πρόσβαση.
- Σε αυτήν την περίπτωση, το ποσοστό των αναγνώσεων (readPercentage) τίθεται πάντα στο 100%.

- Για τη λειτουργία "write":

```

else if (strcmp(basic->operationMode, "write") == 0) {
    /* Όταν write: αν υπάρχει 4ο όρισμα, τότε αυτός είναι ο αριθμός νημάτων, και αν υπάρχει 5ο, αυτό το isRandom */
    if (argc >= 4) {
        advanced->threadCount = atol(argv[3]);
    } else {
        advanced->threadCount = defaults.defaultThreadCount;
    }
    if (advanced->threadCount <= 0) {
        err.code = ERR_INVALID_THREADS;
        strcpy(err.message, "Ο αριθμός νημάτων πρέπει να είναι μεγαλύτερος του 0.");
        handleError(err);
    }
    if (argc >= 5) {
        advanced->isRandom = atoi(argv[4]);
    } else {
        advanced->isRandom = defaults.defaultRandom;
    }
    advanced->readPercentage = 0.0; // Όλες οι εργασίες είναι εγγραφές.
}

```

- Το τρίτο όρισμα (argv[3]) ορίζει τον αριθμό νημάτων, με προεπιλογή το 1.
- Το τέταρτο όρισμα (argv[4]) καθορίζει το είδος της πρόσβασης (τυχαία ή σειριακή), με προεπιλογή τη σειριακή.
- Σε αυτήν την περίπτωση, το ποσοστό αναγνώσεων (readPercentage) τίθεται πάντα στο 0%.

Η παραπάνω διεξοδική και σχολαστική ανάλυση των ορισμάτων έχει στόχο τη διασφάλιση ότι οι παράμετροι που θα χρησιμοποιηθούν στη συνέχεια της προσομοίωσης είναι έγκυρες, λογικές και συμβατές με τις προδιαγραφές της εκάστοτε λειτουργίας. Η συνάρτηση αξιοποιεί την ευέλικτη δομή διαχείρισης σφαλμάτων ([ErrorHandler](#)), παρέχοντας σαφή και κατανοητά μηνύματα στον χρήστη και αποτρέποντας έτσι τη συνέχιση της προσομοίωσης σε περίπτωση εσφαλμένων παραμέτρων.

2.3.2 Πίνακας Παραδειγμάτων

Για να γίνει απόλυτα κατανοητή η λειτουργία της προσομοίωσης, είναι απαραίτητο να εξετάσουμε τόσο τις έγκυρες όσο και τις μη έγκυρες περιπτώσεις των παραμέτρων εισόδου. Στον παρακάτω πίνακα παρουσιάζονται αναλυτικά παραδείγματα για καλύτερη κατανόηση:

Λειτουργία	Παράδειγμα εντολής	Έγκυρη / Μη Έγκυρη	Αιτιολόγηση
Read	./kiwi-bench read 10000 5 1	Έγκυρη	Καθορίζει 10.000 αναγνώσεις με 5 νήματα και τυχαία πρόσβαση.
Write	./kiwi-bench write 5000	Έγκυρη	Εκτελεί 5.000 εγγραφές με 1 νήμα (προεπιλογή) και σειριακή πρόσβαση (προεπιλογή).
Readwrite	./kiwi-bench readwrite 20000 4 0 30	Έγκυρη	20.000 εργασίες, 4 νήματα, σειριακή πρόσβαση, με 30% αναγνώσεις και 70% εγγραφές.
Readwrite	./kiwi-bench readwrite 10000 5 1	Έγκυρη	20.000 εργασίες, 4 νήματα, τυχαία πρόσβαση, με 50% αναγνώσεις και 50% εγγραφές
Readwrite	./kiwi-bench readwrite 10000 1	Μη Έγκυρη	Απαιτούνται τουλάχιστον 2 νήματα για λειτουργία readwrite.
Readwrite	./kiwi-bench readwrite 10000 4 0 120	Μη Έγκυρη	Το ποσοστό αναγνώσεων πρέπει να είναι μεγαλύτερο από 0 και μικρότερο από 100.
Read	./kiwi-bench read -5000	Μη Έγκυρη	Ο αριθμός εργασιών πρέπει να είναι θετικός.
Read	./kiwi-bench write 5000 -2	Μη Έγκυρη	Ο αριθμός νημάτων πρέπει να είναι θετικός.
Write	./kiwi-bench write	Μη Έγκυρη	Μη επαρκής αριθμός ορισμάτων. Χρειάζονται τουλάχιστον 3 εως και 6 ορίσματα.

Η ανάλυση που ακολουθεί στο κεφάλαιο αυτό εξασφαλίζει ότι κάθε παράμετρος έχει αναλυθεί και αιτιολογηθεί πλήρως, ώστε η διαδικασία της προσομοίωσης να γίνεται με ακρίβεια, συνέπεια και αξιοπιστία.

2.3.3 Στιγμιότυπα Σφαλμάτων

1. Λιγότερα νήματα από το ελάχιστο (2) σε readwrite

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

```
Benchmark Error: Ο αριθμός νημάτων πρέπει να είναι μεγαλύτερος του 0. Για λειτουργία readwrite απαιτούνται τουλάχιστον 2 νήματα.
```

2. Λάθος όρια ποσοστού

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench readwrite 10000 4 0 120
```

```
Benchmark Error: Το ποσοστό αναγνώσεων πρέπει να είναι > 0 και μικρότερο του 100..
```

3. Αριθμός εργασιών μικρότερος ή ίσος του μηδενός

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench read -5000
```

```
Benchmark Error: Ο αριθμός εργασιών πρέπει να είναι μεγαλύτερος του 0..
```

4. Αριθμός νημάτων μικρότερος ή ίσος του μηδενός

```
myy601@myy601lab1:~/kiwi/kiwi-source/bench$ ./kiwi-bench write 5000 -2
```

```
Benchmark Error: Ο αριθμός νημάτων πρέπει να είναι μεγαλύτερος του 0. Για λειτουργία readwrite απαιτούνται τουλάχιστον 2 νήματα.
```

5. Ανεπαρκή ορίσματα

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

```
Benchmark Error: Ανεπαρκή ορίσματα. Χρήση: ./kiwi-bench <read | write | readwrite> [<count> [<threads> <random> [<read percentage>]]]
```

2.3.4 Ροή Προγράμματος main()

Η συνάρτηση main αποτελεί το σημείο εκκίνησης της προσομοίωσης και έχει ως βασικό της σκοπό την προετοιμασία και αρχικοποίηση του περιβάλλοντος εκτέλεσης. Η εκτέλεση του προγράμματος χωρίζεται στα εξής σαφή βήματα:

1. Αρχικοποίηση γεννήτριας τυχαίων αριθμών

```
BasicBenchArgs basicArgs;  
AdvancedBenchArgs advArgs;  
  
/* Αρχικοποίηση τυχαίων αριθμών και εκτύπωση header και περιβάλλοντος */  
 srand(time(NULL));
```

Η γεννήτρια τυχαίων αριθμών αρχικοποιείται με την κλήση της συνάρτησης srand(time(NULL)); Η αρχικοποίηση αυτή είναι απαραίτητη για τη χρήση τυχαίων κλειδιών σε λειτουργίες ανάγνωσης ή εγγραφής, όταν επιλεγεί τυχαία πρόσβαση (random).

2. Εκτύπωση γενικών πληροφοριών (header) και πληροφοριών του περιβάλλοντος εκτέλεσης

```
/* Εκτύπωση header και περιβαλλοντικών πληροφοριών */  
_print_header(argc > 2 ? atoi(argv[2]) : 0);  
_print_environment();
```

Η συνάρτηση _print_header καλείται ώστε να εκτυπώσει συνοπτικά δεδομένα σχετικά με το μέγεθος των κλειδιών (Keys), το μέγεθος των τιμών (Values), τον αριθμό των εγγραφών (Entries) και τις εκτιμήσεις των μεγεθών του ευρετηρίου (IndexSize) και των δεδομένων (DataSize).

Αυτή η εκτύπωση δίνει στον χρήστη μια σαφή εικόνα του όγκου δεδομένων που θα διαχειριστεί η προσομοίωση.

3. Περιβάλλον εκτέλεσης

Η συνάρτηση _print_environment εμφανίζει σημαντικές πληροφορίες για το σύστημα στο οποίο εκτελείται η προσομοίωση, όπως:

- Την τρέχουσα ημερομηνία και ώρα.
- Τον αριθμό των διαθέσιμων πυρήνων (cores) της CPU.
- Τον τύπο του επεξεργαστή.
- Το μέγεθος της κρυφής μνήμης (CPU cache).
Η πληροφόρηση αυτή είναι απαραίτητη για τη διασφάλιση της αναπαραγωγιμότητας και της αξιοπιστίας των μετρήσεων της προσομοίωσης.

4. Ανάλυση και επεξεργασία ορισμάτων από τη γραμμή εντολών

```
/* Ανάλυση ορισμάτων από τη γραμμή εντολών */  
parseBenchArgs(argc, argv, &basicArgs, &advArgs);
```

Ακολουθεί η κλήση της συνάρτησης `parseBenchArgs`, η οποία αναλύει τα ορίσματα εισόδου από τη γραμμή εντολών (argv) και τα αποθηκεύει στις δομές δεδομένων:

- [BasicBenchArgs](#) (π.χ. λειτουργία, αριθμός εργασιών).
 - [AdvancedBenchArgs](#) (π.χ. αριθμός νημάτων, ποσοστό αναγνώσεων, τύπος πρόσβασης δεδομένων).
- Αν κάποιο όρισμα είναι μη έγκυρο ή ελλιπές, το πρόγραμμα τερματίζεται με σαφές μήνυμα λάθους.

5. Εκκίνηση του benchmark

```
// Εκκίνηση benchmark
runBenchmark(&basicArgs, &advArgs);

return EXIT_SUCCESS;
```

Τέλος, η συνάρτηση `runBenchmark` καλείται για να ξεκινήσει την κύρια πολυνηματική διαδικασία εκτέλεσης της προσομοίωσης. Αυτή η συνάρτηση υλοποιεί την πραγματική λειτουργικότητα ανάγνωσης ή εγγραφής των δεδομένων, όπως αυτή έχει καθοριστεί από τις παραμέτρους που αναλύθηκαν προηγουμένως.

3. Πολυνηματική Προετοιμασία

Η βασική λογική πίσω από το πρόγραμμα **kiwi.c** είναι η αξιολόγηση της επίδοσης μιας βάσης δεδομένων με πολλαπλά νήματα, τα οποία εκτελούν είτε εγγραφές (**write**), είτε αναγνώσεις (**read**), είτε συνδυασμό των δύο (**readwrite**). Η προσομοίωση χωρίζεται σε δύο βασικά βήματα:

1. **Προετοιμασία και κατανομή εργασιών ([partitionWorkload](#)):**
 - Ο υπολογισμός του αριθμού των νημάτων και των αντίστοιχων εργασιών ανάλογα με τις παραμέτρους εισόδου.
 - Η δίκαιη κατανομή των εργασιών, ώστε να αποφεύγονται περιπτώσεις ανισοκατανομής και να διασφαλίζεται ότι κάθε νήμα εκτελεί σχεδόν ίδιο αριθμό εργασιών.
2. **Εκτέλεση της πολυνηματικής προσομοίωσης ([runBenchmark](#)):**
 - [Δημιουργία των threads.](#)
 - Εκτέλεση εργασιών από κάθε thread.
 - Συγχρονισμός και καταγραφή χρόνων και στατιστικών στοιχείων.
3. **[Εκτύπωση Αποτελεσμάτων Προσομοίωσης](#)**

Η προσομοίωση αρχίζει με την κλήση της `runBenchmark`, η οποία με τη σειρά της καλεί την `partitionWorkload` για τη σωστή προετοιμασία και τον διαμοιρασμό των διεργασιών.

3.1 Ανάλυση Βασικών Δομών

Για να πετύχουμε καταμερισμό εργασιών και προβολή αποτελεσμάτων οργανωμένα χρειαζόμαστε κάποιες δομές για την ευκολία μας κατά την διαδικασία προσομοίωσης. Ακολουθούν κάποιες από τις δομές.

3.1.1 Δομή ThreadBenchArgs

Αποτελεί δομή δεδομένων που περιγράφει την εργασία που θα εκτελέσει κάθε thread.

```

/* Δομή για τα ορίσματα κάθε νηματικής εργασίας */
typedef struct {
    TimerData timer;          // ατομικός χρόνος εκτέλεσης για το συγκεκριμένο thread
    int flagToStartStopTimeForThreadingOperation;
        // = 1 για το πρώτο thread μιας ομάδας (read ή write)
        // = -1 για το τελευταίο thread μιας ομάδας (read ή write)
        // = 0 για όλα τα ενδιάμεσα threads
    long operationsPerThread; // αριθμός εργασιών που θα κάνει το thread
    int isReader;             // αν είναι read-thread: 1, αλλιώς 0
    int isRandom;             // αριθμός κλειδιών που βρέθηκαν (μόνο για read)
} ThreadBenchArgs;

```

- Timer:** Καταγράφει τον χρόνο εκτέλεσης κάθε thread (έναρξη και λήξη).
- flagToStartStopTimeForThreadingOperation:** Καθορίζει ποιο thread ξεκινά και ποιο σταματά τον χρονισμό μιας ομάδας νημάτων.
- operationsPerThread:** Πλήθος εργασιών που θα εκτελέσει το συγκεκριμένο thread.
- isReader:** Αν το thread εκτελεί αναγνώσεις (1) ή εγγραφές (0).
- isRandom:** Προσδιορίζει αν οι προσβάσεις είναι τυχαίες (1) ή σειριακές (0).
- keysFound:** Αριθμός κλειδιών που βρέθηκαν (ισχύει μόνο για αναγνώσεις).

Ειδική αναφορά στο **flagToStartStopTimeForThreadingOperation**:

Τιμή	Περιγραφή
1	Το πρώτο thread της ομάδας, που ξεκινά τον timer της ομάδας.
-1	Το τελευταίο thread της ομάδας, που σταματά τον timer της ομάδας.
0	Ενδιάμεσα threads χωρίς χρονική ευθύνη.

3.1.2 Δομή BenchResults

```

/* Δομή για αποθήκευση των αποτελεσμάτων του καθε operation */
typedef struct {
    ThreadBenchArgs* threadArgs; // πίνακας με structs των threads
    TimerData OperationTimer;    // TimerData για συνολικό χρόνο read ή write ομάδας
    long int keysRetrieved;     // κλειδιά που βρέθηκαν (μόνο για read threads)
    long int totalOperations;   // συνολικός αριθμός εργασιών (όλων των threads μαζί)
    long int totalThreads;      // συνολικά threads που χρησιμοποιήθηκαν
    long double opsPerSecond;   // συνολική απόδοση εργασιών/sec
    long double secPerOp;       // χρόνος ανά εργασία (sec)
    long double secPerThread;   // μέσος χρόνος ανά thread (sec)
} BenchResults;

```

Η δομή αυτή χρησιμοποιείται για την αποθήκευση αποτελεσμάτων συνολικά για μία ομάδα νημάτων (ανάγνωση ή εγγραφή):

- threadArgs:** Πίνακας με τα δεδομένα κάθε thread της ομάδας.
- OperationTimer:** Συνολικός χρόνος εκτέλεσης της ομάδας (read/write).
- keysRetrieved:** Συνολικά κλειδιά που βρέθηκαν από τα read threads.
- totalOperations:** Συνολικές εργασίες που εκτελέστηκαν από όλα τα threads.
- totalThreads:** Συνολικός αριθμός των threads της ομάδας.
- opsPerSecond:** Απόδοση (εργασίες ανά δευτερόλεπτο).
- secPerOp:** Μέσος χρόνος εκτέλεσης ανά εργασία.
- secPerThread:** Μέσος χρόνος εκτέλεσης ανά thread.

3.1.3 Δομή Datas

Η δομή αυτή αποτελεί το γενικό πλαίσιο που ενσωματώνει όλα τα δεδομένα της προσομοίωσης:

```

// Φωλιασμα όλων των δεδομένων για τα test και τα αποτελέσματα
typedef struct {
    TimerData GlobalTimer;           // TimerData συνολικού benchmark (start-end)
    AdvancedBenchArgs *advArgs;     //
    BenchResults *resultsPerOperation; // Array (read/write results)
    int currentThreadIndex;
} Data;

```

- **GlobalTimer:** Ο συνολικός χρόνος εκτέλεσης του benchmark (έναρξη - λήξη).
- **advArgs:** Δείκτης στα προηγμένα ορίσματα της προσομοίωσης.
- **resultsPerOperation:** Πίνακας με BenchResults για κάθε τύπο operation (read, write).
- **currentThreadIndex:** Το index του τρέχοντος thread (χρησιμοποιείται στην εκτέλεση).

3.2 Κατανομή των εργασιών μεταξύ των threads (partitionWorkload)

```
>Data* partitionWorkload(BasicBenchArgs* basic, AdvancedBenchArgs* adv)
```

Η συνάρτηση partitionWorkload είναι υπεύθυνη για την αρχική προετοιμασία και κατανομή των εργασιών μεταξύ των threads.

Η διαδικασία είναι η εξής:

- **Υπολογισμός του πλήθους των threads** ανάλογα με τον τύπο των εργασιών (ανάγνωση, εγγραφή ή συνδυασμός αυτών) και τις παραμέτρους που δόθηκαν:

```

// Καθορίζουμε αν η λειτουργία περιλαμβάνει αναγνώσεις (hasRead) και/ή εγγραφές (hasWrite)
int hasRead = strcmp(basic->operationMode, "write") != 0;
int hasWrite = strcmp(basic->operationMode, "read") != 0;

// Υπολογισμός του αριθμού των τύπων εργασιών (1 ή 2)
int opsCount = hasRead + hasWrite;

// Δεσμεύουμε μνήμη για την αποθήκευση αποτελεσμάτων ανά τύπο εργασίας (read και/ή write)
datas->resultsPerOperation = calloc(opsCount, sizeof(BenchResults));

// Συνολικός αριθμός threads από τα ορίσματα εισόδου
int totalThreads = adv->threadCount;

// Υπολογισμός αρχικού αριθμού threads ανά τύπο (με στρογγυλοποίηση προς τα πάνω για δικαιοσύνη)
int readThreads = hasRead ? (int)(totalThreads * adv->readPercentage / 100.0 + 0.5) : 0;
int writeThreads = totalThreads - readThreads;

```

- **Έλεγχος για οριακές περιπτώσεις:** Για να αποφευχθεί περίπτωση με 0 threads για έναν τύπο εργασιών, γίνονται οι έλεγχοι:

```

// Έλεγχος για αποφυγή περίπτωσης μηδενικού αριθμού threads σε κάποιο τύπο operation
if (hasRead && readThreads == 0) { readThreads = 1; writeThreads--; }
if (hasWrite && writeThreads == 0) { writeThreads = 1; readThreads--; }

```

Παράδειγμα οριακών κατανομών σε πίνακα:

Total Threads	Read%	Read Threads	Write Threads	Σχόλιο
4	25%	1	3	Κανονική κατανομή
4	0.1%	1 (διορθωμένο από 0)	3 (διορθωμένο από 4)	Διόρθωση οριακής περίπτωσης
4	90%	4 (διορθωμένο από 5)	1 (διορθωμένο από 0)	Διόρθωση σε write threads

- **Υπολογισμός εργασιών ανά thread:** Καθορίζεται ο αριθμός των εργασιών ανά thread με δίκαιη κατανομή:

```

// Ορίζουμε αριθμό threads για το READ
readRes->totalThreads = readThreads;

// Συνολικός αριθμός εργασιών READ
readRes->totalOperations = basic->operationCount * adv->readPercentage / 100;

// Δέσμευση μνήμης για τα ορίσματα των read threads
readRes->threadArgs = calloc(readThreads, sizeof(ThreadBenchArgs));

```

- **Διανομή εργασιών στα threads:** Τα πρώτα threads λαμβάνουν μία επιπλέον εργασία μέχρι να εξαντληθεί το υπόλοιπο (extraOps). Αυτό εξασφαλίζει ελάχιστη διαφορά εργασιών μεταξύ threads (δικαιοσύνη κατανομής):

```

// Υπολογισμός του αριθμού εργασιών ανά thread και του υπολοίπου
long opsPerThread = readRes->totalOperations / readThreads;
long extraOps = readRes->totalOperations % readThreads;

// Διανομή εργασιών στα threads (δικαιοσύνη με έξτρα εργασίες στα πρώτα threads)
for (int i=0; i<readThreads; i++) {
    readRes->threadArgs[i].operationsPerThread = opsPerThread + (i < extraOps ? 1 : 0);
    readRes->threadArgs[i].isReader = 1; // είναι thread ανάγνωσης
    readRes->threadArgs[i].isRandom = adv->isRandom;
    // Ορίζουμε ποιο thread ξεκινά και ποιο σταματά τον χρονομετρητή
    readRes->threadArgs[i].flagToStartStopTimeForThreadingOperation =
        (i==0) ? 1 : (i==readThreads-1)? -1 : 0;
}

```

Για παράδειγμα, σε κατανομή 10 εργασιών σε 3 threads έχουμε:

Thread	operationsPerThread
1	4 (3 + 1 extra)
2	3
3	3

Η συνάρτηση partitionWorkload επιστρέφει έναν δείκτη τύπου [Datas](#), ο οποίος είναι η κεντρική δομή της προσομοίωσης.

```

// Δέσμευση μνήμης για τη βασική δομή δεδομένων της προσομοίωσης
Datas *datas = calloc(1, sizeof(Datas));

```

Περιέχει αναλυτικά όλα τα δεδομένα που απαιτούνται για να ξεκινήσει το πολυνηματικό benchmark:

- **resultsPerOperation:**

```

// Διαχείριση για την περίπτωση WRITE
if (hasWrite) {
    // Παίρνουμε δείκτη προς τα αποτελέσματα του WRITE operation
    BenchResults *writeRes = &datas->resultsPerOperation[opIndex];

    // Διαχείριση για την περίπτωση READ
    if (hasRead) {
        // Παίρνουμε δείκτη προς τα αποτελέσματα του READ operation
        BenchResults *readRes = &datas->resultsPerOperation[opIndex++];
    }
}

```

Πρόκειται για έναν πίνακα από δομές [BenchResults](#). Κάθε στοιχείο του πίνακα αντιστοιχεί σε έναν τύπο εργασίας (read/write). Κάθε [BenchResults](#) κρατά πληροφορίες σχετικά με το πλήθος των threads και το σύνολο εργασιών που πρέπει να εκτελεστούν για τον τύπο αυτό.

- **threadArgs:**

Για κάθε thread, η δομή αυτή περιέχει ακριβείς πληροφορίες σχετικά με τις εργασίες που θα εκτελεστούν, την τυχαιότητα πρόσβασης και τη διαχείριση χρόνων.

- **Δίκαιη κατανομή:**

Η κατανομή εργασιών είναι όσο το δυνατόν δικαιότερη, με τα πρώτα threads κάθε ομάδας να λαμβάνουν μία επιπλέον εργασία (αν υπάρχουν), έτσι ώστε να ελαχιστοποιούνται οι διαφορές μεταξύ των threads.

Έτσι, επιστρέφοντας τη δομή [Datas](#), το πρόγραμμα είναι έτοιμο να ξεκινήσει αμέσως την πολυνηματική προσομοίωση με δίκαια κατανεμημένες εργασίες και σαφή δεδομένα για κάθε thread.

3.3 Χρονομέτρηση – Κόστος

Ακολουθεί μια αναλυτική περιγραφή που εξηγεί πώς κρατάμε το χρόνο, ποιες μετρήσεις χρόνου πραγματοποιούμε, πώς τις πετυχαίνουμε με ακρίβεια και πώς διαχειριζόμαστε ειδικές περιπτώσεις όπως η εκτέλεση από ένα μόνο νήμα ή πολλά νήματα παράλληλα

3.3.1 Δομή TimerData

```
// Δομή για αποθήκευση χρόνου
typedef struct {
    struct timespec startTime; // χρόνος έναρξης
    struct timespec endTime; // χρόνος λήξης
    long double totalTime; // Συνολικός χρόνος σε sec
} TimerData;
```

struct timespec: Χρησιμοποιείται για υψηλής ακρίβειας μέτρηση χρόνου σε επίπεδο νανοδευτερολέπτων. Περιέχει τρία πεδία:

- **Struct starttime:** Για το χρόνο έναρξης
- **Struct stoptime:** Για το χρόνο λήξης
- **Total time:** για το αποτέλεσμα της μέτρησης

3.3.2 Τύποι χρόνου μέτρησης

Κάνουμε τρεις τύπους μετρήσεων:

1. Global Timer (συνολικός χρόνος του benchmark)

```
// Εκκινάμε το global timer του benchmark
clock_gettime(CLOCK_MONOTONIC, &datas->GlobalTimer.startTime);
```

- Μετράει τον χρόνο εκτέλεσης ολόκληρου του benchmark από την έναρξη μέχρι τη λήξη όλων των threads και αποθηκεύεται όπως είπαμε στον [Datas struct](#). Θα το δούμε και παρακάτω στην “[runBenchmark \(\)](#)”
- Ξεκινά πριν τη δημιουργία των threads και σταματά μετά το join όλων των threads.

2. Operation Timer (ανά operation, π.χ. READ ή WRITE)

```
// Εκκίνηση χρόνου για ολόκληρη την διαδικασία read αν είναι το πρώτο νήμα
if (tArgs->flagToStartStopTimeForThreadingOperation == 1)
    clock_gettime(CLOCK_MONOTONIC, &datas->resultsPerOperation->OperationTimer.startTime);
```

- Μετράει τον χρόνο εκτέλεσης για κάθε operation χωριστά και αποθηκεύεται στον κατάλληλο [BenchResults struct](#) που έχει φημιουργηθεί για το συγκεκριμένο mode.
- Ξεκινά όταν αρχίσει το πρώτο thread μιας ομάδας (π.χ. των threads ανάγνωσης) και σταματά όταν τελειώσει το τελευταίο thread αυτής της ομάδας.

- Αυτό γίνεται με τη χρήση του flag [flagToStartStopTimeForThreadingOperation](#).

4. Thread Timer (ανά thread)

```
// Ξεκινάμε και το δικό του timer του thread
clock_gettime(CLOCK_MONOTONIC, &tArgs->timer.startTime);
```

- Μετράει τον χρόνο που απαιτείται για την εκτέλεση κάθε thread ξεχωριστά και αποθηκεύεται στον κάθε νήματος το struct ([ThreadBenchArgs](#)) μέσα στον TimerData struct του.
- Ξεκινά όταν αρχίσει η εργασία του thread και σταματά όταν το συγκεκριμένο thread ολοκληρώσει τις εργασίες του.

3.3.3 Ακρίβεια στη μέτρηση

Για να έχουμε τη μέγιστη δυνατή ακρίβεια:

- Χρησιμοποιούμε τη [clock_gettime\(\)](#) με [CLOCK_MONOTONIC](#), καθώς αυτή παρέχει ένα μονοτονικό χρονόμετρο το οποίο δεν επηρεάζεται από αλλαγές συστήματος (π.χ. αλλαγή ώρας).
- Υπολογίζουμε τη διάρκεια με ακρίβεια νανοδευτερολέπτων ως εξής και με την βοηθεία της συνάρτησης [«calculateDuration\(\)»](#):

```
// Σταματά τον timer και υπολογίζει τη διάρκεια
void calculateDuration(TimerData* timer) {
    clock_gettime(CLOCK_MONOTONIC, &timer->endTime);
    timer->totalTime = (timer->endTime.tv_sec - timer->startTime.tv_sec) +
    (timer->endTime.tv_nsec - timer->startTime.tv_nsec) / 1E9;
}
```

- Οι μετρήσεις ξεκινούν και σταματούν ακριβώς στα σημεία όπου αρχίζει και ολοκληρώνεται η εκάστοτε εργασία ή ομάδα εργασιών.

3.3.4 Ειδική περίπτωση μέτρησης

Για να διασφαλίσουμε ακρίβεια και σωστή καταγραφή χρόνου σε ειδικές περιπτώσεις (όπως ένα μόνο νήμα ή πολλά νήματα), χρησιμοποιούμε flags:

Τα flag ορίζονται στην [«partitionWorkload\(\)»](#) συνάρτηση κατά τον διαμερισμό εργασιών και νημάτων.

```
// Ορίζουμε ποιο thread ξεκινά και ποιο σταματά τον χρονομετρητή
writeRes->threadArgs[i].flagToStartStopTimeForThreadingOperation =
    (i==0) ? 1 : (i==writeThreads-1)? -1 : 0;
```

- [flagToStartStopTimeForThreadingOperation](#)

- **1:** Το νήμα αυτό είναι το πρώτο στην ομάδα του και πρέπει να ξεκινήσει τον μετρητή.
- **-1:** Το νήμα αυτό είναι το τελευταίο στην ομάδα του και πρέπει να σταματήσει τον μετρητή.
- **0:** Το νήμα δεν είναι ούτε πρώτο ούτε τελευταίο, οπότε δεν κάνει τίποτα με τον operation timer.

Η υλοποίηση είναι ως εξής (*βλέπε στο κώδικα της `read_test` ή `write_test`*):

- Αν υπάρχει μόνο ένα νήμα, το ίδιο νήμα ξεκινά και σταματά τον timer.

```
if (tArgs->flagToStartStopTimeForThreadingOperation == 1)
    clock_gettime(CLOCK_MONOTONIC, &datas->resultsPerOperation->OperationTimer.startTime);

// Άνην timer του operation αν είναι το τελευταίο thread ή μοναδικό thread
if (tArgs->flagToStartStopTimeForThreadingOperation == -1 || datas->resultsPerOperation->totalThreads == 1)
    clock_gettime(CLOCK_MONOTONIC, &datas->resultsPerOperation->OperationTimer.endTime);
    calculateDuration(&datas->resultsPerOperation->OperationTimer);
```

- Σε περίπτωση πολλών νημάτων, το πρώτο ξεκινά και το τελευταίο σταματά, έτσι ώστε να μετράει τη συνολική διάρκεια της ομάδας των threads και να μην χάνεται ή να επικαλύπτεται χρόνος.

3.3.5 Πώς χειριζόμαστε τις ακραίες περιπτώσεις

A. Ένα νήμα (single-threaded):

Ο operation timer ξεκινά και σταματά από το ίδιο το νήμα, χωρίς παρεμβολές. Έτσι, η ακρίβεια παραμένει υψηλή.

B. Πολλά νήματα (multi-threaded):

Ο operation timer ξεκινά στο πρώτο νήμα και σταματά στο τελευταίο νήμα. Έτσι, μετράμε τον συνολικό χρόνο όλων των παράλληλων εργασιών της ομάδας, διασφαλίζοντας ότι μετράμε την πραγματική διάρκεια του παράλληλου workload.

Γ. Τι πετυχαίνουμε συνολικά;

Με αυτή την προσέγγιση:

- **Υψηλή ακρίβεια** χρόνου χάρη στο CLOCK_MONOTONIC.
- Σωστή μέτρηση του χρόνου **σε όλες τις περιπτώσεις** (single-threaded, multi-threaded).
- **Καθαρότητα** στη μέτρηση κάθε διαφορετικού τύπου operation (read/write).
- **Σαφή** διαχωρισμό των επιμέρους χρόνων (ανά thread, ανά operation, συνολικό benchmark).

3.4 Ροή Εκτέλεσης Διεργασιών `runBenchmark`

1. Προετοιμασία του Workload (partitionWorkload):

```
// Προετοιμασία εργασιών και νημάτων
Datas *datas = partitionWorkload(basic, adv);
```

Εδώ υπολογίζουμε και καθορίζουμε:

- Τι εργασίες θα γίνουν (READ, WRITE, READWRITE).
- Πόσα νήματα (threads) θα χρησιμοποιηθούν για κάθε εργασία.
- Πόσες συνολικές εγγραφές/αναγνώσεις θα εκτελεστούν από κάθε νήμα.

Η συνάρτηση `partitionWorkload` επιστρέφει μια δομή `Data` με όλα τα απαραίτητα στοιχεία και `timers`.

2. Ανοιγμα της βάσης δεδομένων (DB)

Ανοίγουμε την global DB (μία φορά πριν ξεκινήσουν τα νήματα):

```
#define DATAS ("testdb")

/* Global pointer για τη βάση - θα ανοίξει πριν ξεκινήσει το benchmark και θα κλείσει μετά. */
DB* globalDB = NULL;

// Ανοιγμα βάσης δεδομένων
globalDB = db_open(DATAS);
```

Όλα τα threads θα χρησιμοποιούν αυτή τη βάση δεδομένων.

3. Έναρξη Global Timer

```
// Ξεκινάμε το global timer του benchmark
clock_gettime(CLOCK_MONOTONIC, &datas->GlobalTimer.startTime);
```

Ξεκινάμε τον global timer που καταγράφει τον συνολικό χρόνο εκτέλεσης του benchmark.

4. Υπολογισμός συνολικών threads

Υπολογίζουμε πόσα συνολικά threads θα τρέξουν για όλο το benchmark:

```
// Υπολογίζουμε συνολικά πόσα threads έχουμε (read + write)
int totalThreads = 0;
int opsCount = (strcmp(basic->operationMode, "readwrite") == 0) ? 2 : 1;
```

- Αν το mode είναι READWRITE, υπάρχουν 2 ομάδες (READ, WRITE).
- Αν είναι READ ή WRITE μόνο, υπάρχει 1 ομάδα.

5. Δημιουργία threads

Για κάθε operation group (READ ή WRITE):

- Δημιουργούμε δυναμικά τα αντίστοιχα threads και τους δίνουμε ως ορίσματα τις απαραίτητες πληροφορίες.

```
// Δημιουργούμε τα threads
pthread_t *threads = malloc(sizeof(pthread_t) * totalThreads);
Datas *threadDatas = malloc(sizeof(Datas) * totalThreads);

o Κάθε νήμα παίρνει ένα αντίγραφο των κοινών δεδομένων (Datas) αλλά με διαφοροποιημένα πεδία που αφορούν το ίδιο (πχ. index thread).

for (int opIdx = 0; opIdx < opsCount; opIdx++) {
    BenchResults* opResult = &datas->resultsPerOperation[opIdx];

    for (int i = 0; i < opResult->totalThreads; i++) {
        threadDatas[threadCounter] = *datas; // αντιγραφή κοινών δεδομένων (struct copy)
        threadDatas[threadCounter].currentThreadIndex = i; // το index του thread μέσα στην ομάδα του
        threadDatas[threadCounter].resultsPerOperation = opResult; // δικτυη στο τρέχον BenchResults

        if (opResult->threadArgs[i].isReader)
            pthread_create(&threads[threadCounter], NULL, _read_test, &threadDatas[threadCounter]);
        else
            pthread_create(&threads[threadCounter], NULL, _write_test, &threadDatas[threadCounter]);

        threadCounter++;
    }
}
```

- Κάθε νήμα εκτελεί είτε `_read_test` είτε `_write_test`.

6. Εκτέλεση και ολοκλήρωση threads

```
// join threads
for (int i = 0; i < totalThreads; i++)
    pthread_join(threads[i], NULL);
```

Μετά τη δημιουργία, περιμένουμε όλα τα threads να ολοκληρώσουν την εκτέλεσή τους με pthread_join

7. Σταματάμε τον Global Timer

```
// Σταματάμε τον global timer του benchmark
clock_gettime(CLOCK_MONOTONIC, &datas->GlobalTimer.endTime);
calculateDuration(&datas->GlobalTimer);
```

Αφού όλα τα threads έχουν ολοκληρωθεί, σταματάμε τον global timer για να υπολογίσουμε τον συνολικό χρόνο του benchmark.

8. Κλείσιμο βάσης δεδομένων

```
// Κλείσιμο της βάσης
db_close(globalDB);
```

Η βάση κλείνεται αφού ολοκληρωθούν όλα τα threads.

9. Υπολογισμός και εκτύπωση αποτελεσμάτων

Στο τέλος:

```
// Υπολογισμός και εκτύπωση αποτελεσμάτων
for (int opIdx = 0; opIdx < opsCount; opIdx++) {
    computeResults(&datas->resultsPerOperation[opIdx]);
    printResults(&datas->resultsPerOperation[opIdx], basic, adv, datas, opIdx);
}
```

- Καλούμε computeResults για κάθε ομάδα operations (READ/WRITE), η οποία υπολογίζει τις μετρήσεις (ops/sec, sec/op, sec/thread κλπ).
- Εμφανίζουμε τα αποτελέσματα με printResults.

10. Αποδέσμευση πόρων και μνήμης

```
// Ελευθέρωση των thread
free(threads);
free(threadDatas);

// Αποδέσμευση μνήμης των threadArgs
for (int opIdx = 0; opIdx < opsCount; opIdx++)
    free(datas->resultsPerOperation[opIdx].threadArgs);

free(datas->resultsPerOperation);
free(datas);
```

Τέλος, γίνεται αποδέσμευση της μνήμης που δεσμεύσαμε για threads και δομές

3.5 Αποτελέσματα

3.5.1 Υπολογισμός Αποτελεσμάτων (computeResults ())

Η συνάρτηση αυτή έχει ως σκοπό την εξαγωγή βασικών στατιστικών στοιχείων από τα πειράματα που εκτελούνται κατά τη διάρκεια των μετρήσεων, ώστε να αξιολογηθεί η απόδοση της βάσης δεδομένων και των νημάτων.

```
/* computeResults:
   Υπολογίζει τα στατιστικά του benchmark:
   - Global totalCost από το totalTimer του BenchResults.
   - Συνολικό πλήθος keysRetrieved από τα read threads.
   - Μέσους χρόνους (readCost, writeCost) ως μέσος όρος των αντίστοιχων threads.
   - Ops/sec, sec/op, sec/thread.

*/
// Συνάρτηση που υπολογίζει τα αποτελέσματα κάθε BenchResults
void computeResults(BenchResults *res) {
    long double totalTime = res->OperationTimer.totalTime;

    // Αθροισμα κλειδιών και χρόνων threads
    long keysRetrieved = 0;
    long double sumThreadsTime = 0;

    for (int i = 0; i < res->totalThreads; i++) {
        sumThreadsTime += res->threadArgs[i].timer.totalTime;

        // Μόνο αν είναι read προσθέτω keysFound
        if (res->threadArgs[i].isReader)
            keysRetrieved += res->threadArgs[i].keysFound;
    }

    // αποθηκεύω τα αποτελέσματα
    res->keysRetrieved = keysRetrieved;
    res->opsPerSecond = res->totalOperations / totalTime;
    res->secPerOp = totalTime / res->totalOperations;
    res->secPerThread = sumThreadsTime / res->totalThreads;
}
```

Τα αποτελέσματα που υπολογίζονται είναι τα εξής

1. Συνολικός χρόνος εκτέλεσης του κάθε operation

Υπολογίζεται ο συνολικός χρόνος που απαιτήθηκε για την ολοκλήρωση όλων των εργασιών (είτε εγγραφών είτε αναγνώσεων). Η μέτρηση αυτή είναι σημαντική γιατί δείχνει πόσο συνολικά γρήγορα εκτελέστηκε το σύνολο των αιτήσεων που ζητήθηκαν.

2. Αριθμός κλειδιών που βρέθηκαν (μόνο στις αναγνώσεις)

Για τις αναγνώσεις, υπολογίζεται πόσα κλειδιά εντοπίστηκαν επιτυχώς στη βάση δεδομένων. Το αποτέλεσμα αυτό είναι χρήσιμο για την επιβεβαίωση της ορθότητας της ανάγνωσης και την αξιοπιστία των μετρήσεων, ώστε να ελέγχουμε εάν όλες οι αναζητήσεις επιστρέφουν ορθά αποτελέσματα.

3. Συνολική απόδοση (Operations per second - Ops/sec)

Αυτή η μέτρηση δείχνει πόσες εργασίες (εγγραφές ή αναγνώσεις) πραγματοποιούνται ανά δευτερόλεπτο. Πρόκειται για έναν κρίσιμο δείκτη απόδοσης (throughput), ο οποίος αντικατοπτρίζει την ταχύτητα με την οποία η βάση δεδομένων επεξεργάζεται αιτήματα. Υψηλή τιμή Ops/sec σημαίνει υψηλότερη απόδοση.

4. Μέσος χρόνος ανά εργασία (Seconds per operation - sec/op)

Ο χρόνος αυτός υπολογίζει κατά μέσο όρο πόσο χρόνο χρειάζεται κάθε μεμονωμένη εργασία (εγγραφή ή ανάγνωση) για να ολοκληρωθεί. Πρόκειται για ένα σημαντικό στοιχείο για να εκτιμήσουμε το latency, δηλαδή το πόσο γρήγορα μπορεί η βάση να απαντά σε κάθε αίτημα.

5. Μέσος χρόνος ανά thread (Seconds per thread - sec/thread)

Ο μέσος χρόνος ανά νήμα υπολογίζει πόσο χρόνο χρειάστηκε κάθε νήμα κατά μέσο όρο για να εκτελέσει τις εργασίες που του ανατέθηκαν. Αυτό το μέγεθος μας βοηθά να δούμε πώς κατανεμήθηκε η εργασία στα threads και αν υπήρξε ισορροπημένη κατανομή των πόρων.

3.5.2 Εκτύπωση Αποτελεσμάτων (printResults())

Η συνάρτηση αυτή είναι υπεύθυνη για τη σαφή και κατανοητή παρουσίαση των αποτελεσμάτων και των παραμέτρων που χρησιμοποιήθηκαν κατά το benchmark.

Ειδικότερα εμφανίζει:

A. Παραμέτρους εκτέλεσης (αρχικά αποτελέσματα)

```
// Συνάρτηση που εμφανίζει τα αποτελέσματα στην οθόνη με συνολικό χρόνο benchmark
void printResults(BenchResults *res, BasicBenchArgs *basicArgs, AdvancedBenchArgs *advArgs, Datas *datas, int opIdx
{
    if(opIdx == 0){
        printf("\n===== Ορίσματα benchmark =====\n");
        printf("Λειτουργία: %s\n", basicArgs->operationMode);
        printf("Συνολικές εργασίες: %ld\n", basicArgs->operationCount);
        printf("Αριθμός νημάτων: %ld\n", advArgs->threadCount);
        printf("Τυχαία πρόσβαση: %s\n", advArgs->isRandom ? "Ναι" : "Όχι");
        if (strcmp(basicArgs->operationMode, "readwrite") == 0) {
            printf("Ποσοστό αναγνώσεων: %.2f%\n", advArgs->readPercentage);
        }
    }

    // Εκτύπωση συνολικού χρόνου benchmark (global timer)
    printf("Συνολικός χρόνος Benchmark: %.4Lf sec\n", datas->GlobalTimer.totalTime);
}
```

- Λειτουργία:** Τύπος του benchmark που εκτελείται (read, write, readwrite).
- Συνολικές εργασίες:** Πλήθος των εργασιών (operations) που ζητήθηκαν συνολικά.
- Αριθμός νημάτων:** Το πλήθος των νημάτων που χρησιμοποιήθηκαν για τη μέτρηση.
- Τυχαία πρόσβαση:** Αν οι εγγραφές/αναγνώσεις είναι τυχαίες ή σειριακές.
- Ποσοστό αναγνώσεων:** Εμφανίζεται μόνο στο readwrite mode και δείχνει ποιο είναι το ποσοστό των εργασιών που είναι αναγνώσεις έναντι των εγγραφών.
- Συνολικός χρόνος Benchmark:** Συνολικός χρόνος που χρειάστηκε για την πλήρη εκτέλεση όλου του benchmark, από την έναρξη μέχρι και την ολοκλήρωση όλων των threads.

B. Αποτελέσματα ανά τύπο εργασίας (READ/WRITE)

```
printf("\n===== Αποτελέσματα benchmark %s =====\n", res->threadArgs[0].isReader ? "READ" : "WRITE");
printf("Threads που χρησιμοποιήθηκαν: %ld\n", res->totalThreads);
printf("Συνολικός αριθμός εργασιών: %ld\n", res->totalOperations);
printf("Συνολικός χρόνος operation: %.4Lf sec\n", res->OperationTimer.totalTime);
printf("Συνολική απόδοση (ops/sec): %.2Lf\n", res->opsPerSecond);
printf("Μέσος χρόνος ανά εργασία (sec/op): %.6Lf sec\n", res->secPerOp);
printf("Μέσος χρόνος ανά thread (sec/thread): %.6Lf sec\n", res->secPerThread);

if (res->threadArgs[0].isReader) {
    printf("Συνολικά κλειδιά που βρέθηκαν: %ld\n", res->keysRetrieved);
}
```

- Threads που χρησιμοποιήθηκαν:** Το πλήθος των νημάτων που εκτέλεσαν τις συγκεκριμένες εργασίες (αναγνώσεις ή εγγραφές).
- Συνολικός αριθμός εργασιών:** Πόσες συνολικά εργασίες εκτελέστηκαν από αυτά τα threads.
- Συνολικός χρόνος operation:** Ο συνολικός χρόνος που χρειάστηκε για να ολοκληρωθούν όλες οι εργασίες του συγκεκριμένου τύπου (ανάγνωση ή εγγραφή).

- **Συνολική απόδοση (ops/sec):** Πόσες εργασίες ολοκληρώθηκαν ανά δευτερόλεπτο (μέτρο ταχύτητας).
- **Μέσος χρόνος ανά εργασία (sec/op):** Ο μέσος χρόνος που απαιτήθηκε για την ολοκλήρωση κάθε εργασίας.
- **Μέσος χρόνος ανά thread (sec/thread):** Ο μέσος χρόνος που χρειάστηκε κάθε νήμα ξεχωριστά για να ολοκληρώσει το μερίδιο των εργασιών του.
- **Συνολικά κλειδιά που βρέθηκαν:** (μόνο για read) Το πλήθος των επιτυχών αναζητήσεων στη βάση.

Γ. Στιγμιότυπα Επιτυχών Εκτυπώσεων Μετρικών και Αποτελεσμάτων

- **Write operation mode:**

```
=====
===== Ορίσματα benchmark =====
Λειτουργία: write
Συνολικές εργασίες: 100000
Αριθμός νημάτων: 20
Τυχαία πρόσβαση: Όχι
Συνολικός χρόνος Benchmark: 1.8811 sec

===== Αποτελέσματα benchmark WRITE =====
Threads που χρησιμοποιήθηκαν: 20
Συνολικός αριθμός εργασιών: 100000
Συνολικός χρόνος operation: 1.8653 sec
Συνολική απόδοση (ops/sec): 53611.90
Μέσος χρόνος ανά εργασία (sec/op): 0.000019 sec
Μέσος χρόνος ανά thread (sec/thread): 1.761808 sec
```

- **Read operation mode:**

```
=====
===== Ορίσματα benchmark =====
Λειτουργία: read
Συνολικές εργασίες: 100000
Αριθμός νημάτων: 20
Τυχαία πρόσβαση: Όχι
Συνολικός χρόνος Benchmark: 1.1749 sec

===== Αποτελέσματα benchmark READ =====
Threads που χρησιμοποιήθηκαν: 20
Συνολικός αριθμός εργασιών: 100000
Συνολικός χρόνος operation: 1.1141 sec
Συνολική απόδοση (ops/sec): 89755.92
Μέσος χρόνος ανά εργασία (sec/op): 0.000011 sec
Μέσος χρόνος ανά thread (sec/thread): 1.123723 sec
Συνολικά κλειδιά που βρέθηκαν: 100000
```

- **Readwrite operation mode:**

```
=====
===== Ορίσματα benchmark =====
Λειτουργία: readwrite
Συνολικές εργασίες: 100000
Αριθμός νημάτων: 40
Τυχαία πρόσβαση: Όχι
Ποσοστό αναγνώσεων: 42.00%
Συνολικός χρόνος Benchmark: 1.5366 sec

===== Αποτελέσματα benchmark READ =====
Threads που χρησιμοποιήθηκαν: 17
Συνολικός αριθμός εργασιών: 42000
Συνολικός χρόνος operation: 1.1450 sec
Συνολική απόδοση (ops/sec): 36682.31
Μέσος χρόνος ανά εργασία (sec/op): 0.000027 sec
Μέσος χρόνος ανά thread (sec/thread): 1.242019 sec
Συνολικά κλειδιά που βρέθηκαν: 42000

===== Αποτελέσματα benchmark WRITE =====
Threads που χρησιμοποιήθηκαν: 23
Συνολικός αριθμός εργασιών: 58000
Συνολικός χρόνος operation: 1.3865 sec
Συνολική απόδοση (ops/sec): 41831.55
Μέσος χρόνος ανά εργασία (sec/op): 0.000024 sec
Μέσος χρόνος ανά thread (sec/thread): 1.458022 sec
```

4 Σύστημα και Τροποποιήσεις

Παρακάτω παρουσιάζεται μια τεχνική ανάλυση σε επίπεδο παραγράφων, καθώς και αναφορά στο γιατί σε περιβάλλον με 2GB RAM και 2 πυρήνες (2 cores) παρατηρούνται σφάλματα (segmentation faults, bus errors, aborts), ενώ με 1GB/1core δεν εμφανίζονται τα ίδια προβλήματα. Στο τέλος, περιγράφεται και ο ρόλος του makefile και των αρχείων που ορίσαμε για τις κλειδαριές.

4.1 Προδιαγραφές Συστήματος

1. Αύξηση του ταυτόχρονου παράλληλου φορτίου

Σε ένα σύστημα με **περισσότερους πυρήνες**, ο κώδικας τρέχει πραγματικά σε παράλληλα threads. Με 1 πυρήνα, ο χρόνος εκτέλεσης των threads διαμοιράζεται σειριακά, με αποτέλεσμα να εμφανίζονται λιγότερο έντονα ή και καθόλου τα προβλήματα συγχρονισμού και οι παθολογίες (race conditions).

- Όταν έχουμε 2 πυρήνες, πολλά threads εκτελούνται ταυτόχρονα, οπότε οι ατέλειες στον κώδικα (π.χ. ανεπαρκές κλείδωμα / ξεκλείδωμα) αποκαλύπτονται πολύ πιο έντονα.

2. Race Conditions και ασταθής συμπεριφορά

Αν δεν έχει υλοποιηθεί ορθά ο αμοιβαίος αποκλεισμός, τότε όταν δύο ή περισσότερα threads προσπέλαζουν ή τροποποιούν κοινή μνήμη ταυτόχρονα, προκαλούν ασυνεπή κατάσταση των δεδομένων, η οποία οδηγεί σε:

- **Segmentation Fault** (προσπέλαση μη έγκυρης διεύθυνσης μνήμης),
- **Bus Error** (εσφαλμένη στοίχιση ή πρόσβαση στη μνήμη),
- **Aborts** (σήματα λάθους από τον πυρήνα).

Με 1 πυρήνα και χαμηλότερο concurrency, τέτοιες καταστάσεις μπορεί απλώς να μην προκύψουν συχνά.

3. Αύξηση Μεγέθους Δεδομένων σε 2GB

Μπορεί η αυξημένη μνήμη (2GB) να ενθαρρύνει τη δημιουργία περισσότερων δομών / μεγαλύτερων κρυφών buffers και συναφώς να πολλαπλασιάσει τη χρήση shared resources. Αν ο κώδικας δεν κλειδώνει σχολαστικά αυτές τις περιοχές, οι πιθανότητες για σφάλμα πολλαπλασιάζονται.

4.2 Αρχεία Κλειδαριών και Makefile

1. rwlocker.h και rwlocker.c



- Στο header (.h) αρχείο ορίζονται οι **δομές** (struct) και **πρωτότυπα συναρτήσεων** (function prototypes).
- Στο .c αρχείο υλοποιούνται οι αντίστοιχες συναρτήσεις, ώστε να μπορούν να συμπεριληφθούν (με #include) στο κώδικα μας στα sst αρχεία.

- Η οργάνωση αυτή διατηρεί καθαρή τη διαχείριση του κώδικα και βοηθά στην επαναχρησιμοποίηση του μηχανισμού κλειδώματος και σε άλλα έργα.

Στο makefile του φακέλου engine ορίσαμε το object “rwlocker.o” για τα αρχεία των κλειδαριών [rwlocker.h](#) και [rwlocker.c](#)

2. makefile

```
include ../defs.mk

LIBINDEXER_OBJ = \
    db.o \
    memtable.o \
    indexer.o \
    sst.o \
    sst_builder.o \
    sst_loader.o \
    sst_block_builder.o \
    hash.o \
    bloom_builder.o \
    merger.o \
    compaction.o \
    skiplist.o \
    buffer.o \
    arena.o \
    utils.o \
    crc32.o \
    file.o \
    heap.o \
    vector.o \
    log.o \
    lru.o \
    rwlocker.o # <-- Προσθέσαμε εδώ το νέο object αρχείο για τις κλειδαριές
LIBINDEXER = libindexer.a
```

2. Sst.h

```
#ifndef __SST_H__
#define __SST_H__

#include <pthread.h>
#include "indexer.h"
#include "skiplist.h"
#include "memtable.h"
#include "sst_loader.h"
#include "sst_builder.h"
#include "variant.h"
#include "vector.h"
#include "file.h"
#include "lru.h"
#include "rwlocker.h"
```

4.3 Compile

```
CC lru.o
CC rwlocker.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-variable bench.c kiwi.c -L ../engine -lindexer -lpthread -ls
nappy -o kiwi-bench
make[1]: Leaving directory '/home/myy601/kiwi/kiwi-source/bench'
```

5. Αμοιβαίος Αποκλεισμός

5.1 Γενικά

Η έννοια του **αμοιβαίου αποκλεισμού (Mutual Exclusion)** είναι θεμελιώδης στην παράλληλη και κατανεμημένη επεξεργασία. Αφορά τον μηχανισμό με τον οποίο διασφαλίζεται ότι μόνο μία διεργασία ή ένα νήμα μπορεί να έχει πρόσβαση σε μια κοινόχρηστη περιοχή ή πόρο, τη δεδομένη χρονική στιγμή. Ο στόχος του αμοιβαίου αποκλεισμού είναι να αποφευγθούν καταστάσεις **race conditions**, οι οποίες μπορεί να οδηγήσουν σε απροσδιόριστη ή λανθασμένη συμπεριφορά των προγραμμάτων.

5.1.1 Αναγκαιότητα του αμοιβαίου αποκλεισμού

Κατά την ανάπτυξη πολυνηματικών (multithreaded) εφαρμογών, οι διεργασίες (ή threads) συχνά έχουν ανάγκη πρόσβασης σε κοινούς πόρους, όπως δομές δεδομένων, αρχεία, μεταβλητές κατάστασης κ.λπ. Χωρίς την κατάλληλη διαχείριση πρόσβασης, είναι πιθανό να δημιουργηθεί μια κατάσταση όπου δύο ή περισσότερα νήματα να προσπαθούν ταυτόχρονα να μεταβάλλουν τα δεδομένα αυτά, με αποτέλεσμα να έχουμε ανακριβή αποτελέσματα ή ακόμη και κρασαρίσματα (segmentation faults).

Έτσι, η σωστή υλοποίηση του μηχανισμού κλειδώματος είναι απολύτως απαραίτητη, καθώς διασφαλίζει:

- **Ορθότητα (Correctness):** Να μην υπάρχει αλλοίωση των δεδομένων.
- **Συνέπεια (Consistency):** Οι κοινές δομές δεδομένων να παραμένουν πάντα σε έγκυρη κατάσταση.
- **Αποφυγή Αδιεξόδων (Deadlock Avoidance):** Να αποφεύγονται καταστάσεις όπου οι διεργασίες περιμένουν επ' άπειρον η μία την άλλη.

5.1.2 Τύποι κλειδώματος και στρατηγικές

Υπάρχουν διάφοροι τύποι κλειδώματος που εφαρμόζονται ανάλογα με το σενάριο χρήσης και την απαίτηση απόδοσης του συστήματος:

- **Απλό Mutex (Mutual Exclusion Lock):** Εξασφαλίζει αποκλειστική πρόσβαση για ένα μόνο νήμα κάθε φορά, χωρίς καμία διάκριση.
- **Κλείδωμα Ανάγνωσης-Εγγραφής (Reader-Writer Lock):** Επιτρέπει την ταυτόχρονη πρόσβαση πολλών αναγνωστών, αλλά μόνο ενός εγγραφέα κάθε φορά.
- **Κλείδωμα Προτεραιότητας (Priority Lock):** Ειδική περίπτωση του reader-writer lock, που δίνει προτεραιότητα είτε στους αναγνώστες είτε στους εγγραφείς.

Στην περίπτωσή μας, έχουμε υλοποιήσει έναν **μηχανισμό reader-writer lock με προτεραιότητα στον εγγραφέα** (writer-priority lock). Δηλαδή, πολλοί αναγνώστες μπορούν να διαβάζουν παράλληλα, αλλά μόλις εμφανιστεί ένας εγγραφέας, έχει προτεραιότητα και οι επόμενοι αναγνώστες θα περιμένουν μέχρι να ολοκληρωθεί η εγγραφή. Αυτό γίνεται για να αποφύγουμε τη λεγόμενη **starvation** (λιμοκτονία) του εγγραφέα, όπου οι συνεχείς αναγνώσεις εμποδίζουν τη διαδικασία εγγραφής να συμβεί.

5.2 Ο δικός μας μηχανισμός (RWLocker)

Η σχεδίαση του συγκεκριμένου μηχανισμού κλειδώματος έχει τους εξής στόχους:

- **Παραλληλισμός στις αναγνώσεις:** Να επιτρέπει τη μέγιστη δυνατή παράλληλη πρόσβαση για αναγνώστες, χωρίς άσκοπη αναμονή.
- **Προτεραιότητα στις εγγραφές:** Να εξασφαλίζεται ότι οι εγγραφείς δεν περιμένουν υπερβολικά και δεν συμβαίνει starvation.
- **Ελαχιστοποίηση χρόνων αναμονής (Waiting Time):** Να περιορίσουμε τους χρόνους που ένα νήμα περιμένει, αυξάνοντας τη συνολική απόδοση.

Ο τρόπος σκέψης μας είναι να διαχωρίσουμε την πρόσβαση στις δομές δεδομένων με βάση τη λειτουργία (ανάγνωση ή εγγραφή). Οι αναγνώστες μοιράζονται την πρόσβαση όσο δεν υπάρχει

εγγραφή σε εξέλιξη, ενώ ο εγγραφέας, όταν απαιτείται, αποκτά αποκλειστική πρόσβαση. Η χρήση condition variables εξασφαλίζει ότι ένα νήμα που περιμένει δεν σπαταλάει πόρους (busy-waiting), αλλά «κοιμάται» μέχρι να ενεργοποιηθεί εκ νέου από κάποιο άλλο νήμα

5.2.2 Δομή Κλειδαριάς RWLocker

Ο μηχανισμός μας (RWLocker) βασίζεται στην εξής δομή που βρίσκεται στο αρχείο `rwlocker.h`:

```
typedef struct {
    pthread_mutex_t readers_mutex;      // Mutex για προστασία του μετρητή readers_count
    pthread_mutex_t writers_mutex;      // Mutex για την αποκλειστική πρόσβαση από writer
    pthread_cond_t readers_finished_cond; // Condition variable για αναμονή ολοκλήρωσης των αναγνωστών
    int readers_count;                // Αριθμός αναγνωστών που είναι ενεργοί
    int writer_active;                // Ένδειξη αν υπάρχει writer ενεργός (0/1)
} RWLocker;
```

- **Mutex για τους αναγνώστες (readers_mutex):** Προστατεύει τον μετρητή των ενεργών αναγνωστών (`readers_count`).
- **Mutex για τους εγγραφείς (writers_mutex):** Εξασφαλίζει την αποκλειστική πρόσβαση του εγγραφέα στα δεδομένα.
- **Condition Variable (readers_finished_cond):** Χρησιμοποιείται ώστε οι εγγραφείς να αναμένουν την ολοκλήρωση των αναγνωστών.
- **Μετρητής αναγνωστών (readers_count):** Καταγράφει πόσοι αναγνώστες είναι ενεργοί τη δεδομένη στιγμή.
- **Ένδειξη ενεργού εγγραφέα (writer_active):** Υποδεικνύει εάν υπάρχει αυτή τη στιγμή ενεργή διαδικασία εγγραφής.

Η δομή αυτή αρχικοποιείται μέσα στο `sst.h` αρχείο στην δομή `sst`

```
typedef struct _sst {
    char basedir[MAX_FILENAME];
    unsigned under_compaction:1;
    uint32_t last_id;
    uint32_t file_count;
    File* manifest;
    int comp_level;
    double comp_score;
    RWLocker rlock; // κλειδαριά
    Vector* targets;
    LRU* cache;
}

#ifndef BACKGROUND_MERGE
    MemTable* immutable;
    SkipList* immutable_list;
    pthread_mutex_t immutable_lock;

```

5.2.3 Ανάλυση αποκλεισμού και ταυτοχρονισμού

Στις πολυνηματικές εφαρμογές, συχνά απαιτείται η ταυτόχρονη πρόσβαση πολλών νημάτων σε κοινές δομές δεδομένων. Ενώ η ταυτόχρονη ανάγνωση μπορεί συχνά να πραγματοποιείται με ασφάλεια, η εκτέλεση γραφής (π.χ. εισαγωγές/διαγραφές σε μια βάση δεδομένων) πρέπει να γίνεται αποκλειστικά, για να αποφεύγονται ασυνεπείς καταστάσεις (race conditions). Η κλειδαριά αναγνωστών-εγγραφέων (Reader-Writer Lock) που υλοποιείται στο αρχείο `rwlocker.h` συνδυάζει αυτά τα δύο σενάρια: αφενός επιτρέπει πολλούς αναγνώστες να αποκτούν ταυτόχρονα πρόσβαση, αφετέρου επιβάλλει αμοιβαίο αποκλεισμό όταν ένας γράφων (writer) πρέπει να επέμβει στα δεδομένα.

Με αυτόν τον τρόπο, επιτυγχάνεται υψηλός βαθμός παραλληλίας για τις αναγνώσεις και αποφεύγονται λάθη κατά τη διάρκεια της εγγραφής. Ο συγκεκριμένος μηχανισμός βασίζεται

σε δύο διαφορετικά *mutexes* και ένα *condition variable* που επιτρέπουν τον έλεγχο της ροής των threads.

```
// prototypes
void rwlock_init(RWLocker *lock);
void rwlock_destroy(RWLocker *lock);
void rwlock_reader_lock(RWLocker *lock);
void rwlock_reader_unlock(RWLocker *lock);
void rwlock_writer_lock(RWLocker *lock);
void rwlock_writer_unlock(RWLocker *lock);
```

1. Αρχικοποίηση (Initialization)

```
// Αρχικοποίει το αντικείμενο RWLocker:
// 1) Κατασκευάζει τους mutexes για τους αναγνώστες/γράφοντες.
// 2) Κατασκευάζει το condition variable για την επικοινωνία.
void rwlock_init(RWLocker *lock) {
    pthread_mutex_init(&lock->readers_mutex, NULL);           // Αρχικοποίηση mutex για τον μετρητή αναγνωστών
    pthread_mutex_init(&lock->writers_mutex, NULL);          // Αρχικοποίηση mutex για τους εγγραφείς
    pthread_cond_init(&lock->readers_finished_cond, NULL); // Αρχικοποίηση condition variable
    lock->readers_count = 0;                                // Μηδέν ενεργοί αναγνώστες
    lock->writer_active = 0;                               // Κανένας ενεργός εγγραφέας
}
```

Η κλειδαριά αρχικοποιείται μέσω της συνάρτησης `rwlock_init`, η οποία δημιουργεί:

- Ένα mutex για τη διαχείριση των αναγνωστών (`readers_mutex`),
- Ένα mutex για τον αποκλειστικό έλεγχο των εγγραφέων (`writers_mutex`),
- Ένα condition variable (`readers_finished_cond`) για τη συγχρονισμένη αναμονή μεταξύ αναγνωστών και εγγραφέων.
Παράλληλα, οι εσωτερικές μετρητικές μεταβλητές (`readers_count`, `writer_active`) τίθενται σε αρχικές τιμές (0).

Η αρχικοποίηση του γίνεται στο αρχείο “sst.c” στην συνάρτηση “sst_new”

```
SST* sst_new(const char* basedir, uint64_t cache_size)
{
    SST* self = (SST*)malloc(sizeof(SST));

    strncpy(self->basedir, basedir, sizeof(self->basedir));
    strncat(self->basedir, "/si", MAX_FILENAME);
    mkdirp(self->basedir);

    ...
    rwlock_init(&self->rwlock); // <-- αρχικοποίηση του rwlock
    _read_manifest(self);

    return self;
}
```

2. Λειτουργία Αναγνώστη (Reader Lock/Unlock)

```
// Λειτουργία όταν κάποιος αναγνώστης (reader) θέλει να αποκτήσει πρόσβαση:
// 1) Κατεβάνουμε το readers_mutex για να προστατεύουμε το readers_count.
// 2) Αν υπάρχει ενεργός writer (lock->writer_active == 1), περιμένουμε το condition (readers_finished_cond).
// 3) Όταν δεν υπάρχει ούτε writer, αυξάνουμε το readers_count κατά 1.
// 4) Σεκελιδώνουμε το readers_mutex.

void rwlock_reader_lock(RWLocker *lock) {
    pthread_mutex_lock(&lock->readers_mutex); // Προστασία μεταβλητής readers_count
    while (lock->writer_active) {           // Αν υπάρχει εγγραφέας, περιμένει
        pthread_cond_wait(&lock->readers_finished_cond, &lock->readers_mutex);
    }
    lock->readers_count++;                // Αυξάνει τον αριθμό αναγνωστών
    pthread_mutex_unlock(&lock->readers_mutex); // Αποδέσμευση mutex
}

// Απελευθερώνει την πρόσβαση του αναγνώστη:
// 1) Κατεβάνουμε ξανά το readers_mutex για να μειώσουμε το readers_count.
// 2) Αν πέσει στο μηδέν, ειδοποιούμε signal έναν πιθανό writer που περιμένει.
// 3) Σεκελιδώνουμε.
void rwlock_reader_unlock(RWLocker *lock) {
    pthread_mutex_lock(&lock->readers_mutex);
    lock->readers_count--;                  // Μειώνει τους ενεργούς αναγνώστες
    if (lock->readers_count == 0) {          // Αν δεν υπάρχουν άλλοι αναγνώστες
        pthread_cond_signal(&lock->readers_finished_cond); // Ειδοποιεί έναν εγγραφέα
    }
    pthread_mutex_unlock(&lock->readers_mutex);
}
```

- Κατά την είσοδο ενός αναγνώστη (συνάρτηση rwlock_reader_lock), ελέγχεται πρώτα αν υπάρχει ενεργός εγγραφέας (writer_active == 1). Αν ναι, ο αναγνώστης μπαίνει σε αναμονή μέσω του condition variable (readers_finished_cond).
- Μόλις ο εγγραφέας απελευθερώσει την κλειδαριά, οι αναγνώστες συνεχίζουν, αυξάνοντας το readers_count.
- Όταν ένας αναγνώστης τελειώνει (συνάρτηση rwlock_reader_unlock), μειώνει το readers_count. Αν αυτό γίνει μηδενικό, ειδοποιεί τυχόν εγγραφέα που περιμένει (signal).

3. Λειτουργία Εγγραφέα (Writer Lock/Unlock)

```
// Απελευθερώνει την πρόσβαση του αναγνώστη:
// 1) Κλειδώνουμε ξανά το readers_mutex για να μειώσουμε το readers_count.
// 2) Αν πέσει στο μηδέν, ειδοποιούμε με signal έναν πιθανό writer που περιμένει.
// 3) Επελευθερώνουμε.
void rwlock_writer_lock(RWLocker *lock) {
    pthread_mutex_lock(&lock->writers_mutex); // Αποκλειστικό mutex για εγγραφέα
    pthread_mutex_lock(&lock->readers_mutex); // Προστατεύουμε readers_count
    lock->writer_active = 1; // Δηλώνουμε ενεργό εγγραφέα
    while (lock->readers_count > 0) { // Αναμονή μέχρι να τελειώσουν οι αναγνώστες
        pthread_cond_wait(&lock->readers_finished_cond, &lock->readers_mutex);
    }
    pthread_mutex_unlock(&lock->readers_mutex); // Αποδέσμευση readers mutex
}

// Όταν ο γράφων τελειώσει:
// 1) Πάρει το readers_mutex για να μη συμπέσει με νέους αναγνώστες.
// 2) Θέτει writer_active = 0, κάνει broadcast για να ξυμνήσει οναγνώστες ή άλλους γράφοντες.
// 3) Επελευθερώνει readers_mutex και writers_mutex.
void rwlock_writer_unlock(RWLocker *lock) {
    pthread_mutex_lock(&lock->readers_mutex); // Αποκλειστικό mutex για αναγνώστες
    lock->writer_active = 0; // Δηλώνουμε ανενεργό εγγραφέα
    pthread_cond_broadcast(&lock->readers_finished_cond); // Εκκίνηση όλων των αναγνωστών broadcast
    pthread_mutex_unlock(&lock->readers_mutex); // Αποδέσμευση readers mutex
    pthread_mutex_unlock(&lock->writers_mutex); // Αποδέσμευση writers mutex
}
```

- Όταν ένας εγγραφέας ζητήσει πρόσβαση (συνάρτηση rwlock_writer_lock), κλειδώνει πρώτα τον writers_mutex (ώστε να μην υπάρχει δεύτερος εγγραφέας ταυτόχρονα) και στη συνέχεια τον readers_mutex (για να εμποδίσει νέους αναγνώστες).
- Θέτει writer_active = 1 και αναμένει με condition wait όσο readers_count > 0, ώστε να τελειώσουν οι ήδη ενεργοί αναγνώστες.
- Απελευθερώνει τον readers_mutex για να πραγματοποιήσει την εγγραφή.
- Όταν τελειώσει (συνάρτηση rwlock_writer_unlock), κλειδώνει πάλι τον readers_mutex, θέτει writer_active = 0 και κάνει pthread_cond_broadcast για να επιτραπεί σε αναγνώστες ή άλλους εγγραφείς να συνεχίσουν.

4. Καταστροφή (Destruction)

```
// Καταστρέφει τους πόρους (mutexes και cond) όταν δεν χρειάζονται πλέον.
void rwlock_destroy(RWLocker *lock) {
    pthread_mutex_destroy(&lock->readers_mutex); // Καταστροφή mutex αναγνωστών
    pthread_mutex_destroy(&lock->writers_mutex); // Καταστροφή mutex εγγραφέων
    pthread_cond_destroy(&lock->readers_finished_cond); // Καταστροφή condition variable
}
```

Με τη συνάρτηση rwlock_destroy, τα mutexes και το condition variable καταστρέφονται, απελευθερώνοντας τους πόρους του συστήματος. Προϋποθέτει ότι δεν υπάρχουν ενεργά threads που βασίζονται στην ίδια κλειδαριά. Η καταστροφή τους γίνεται στο αρχείο “sst.c” στην συνάρτηση “sst_free”

```
void sst_free(SST* self)
{
#ifdef BACKGROUND_MERGE
    INFO("Sending termination message to the detached thread")

    pthread_mutex_lock(&self->cv_lock);
    self->merge_state |= MERGE_STATUS_EXIT;
    pthread_cond_signal(&self->cv);
    pthread_mutex_unlock(&self->cv_lock);
}
```

```

    vector_free(self->targets);
    lru_free(self->cache);
    rwlock_destroy(&self->rwlock); // <-- καταστροφή του rwlock
    free(self);
}

```

Α Πίνακας Συναρτήσεων

Συνάρτηση	Ρόλος και λειτουργία
rwlock_init	Αρχικοποιεί mutexes, condition variables, μετρητές.
rwlock_destroy	Καταστρέφει mutexes και condition variables.
rwlock_reader_lock	Αναμονή αν υπάρχει ενεργός writer, αλλιώς ανάγνωση.
rwlock_reader_unlock	Ενημερώνει και ξυπνάει τυχόν writers που περιμένουν.
rwlock_writer_lock	Αποκλειστική πρόσβαση: αναμένει μέχρι να τελειώσουν οι αναγνώστες.
rwlock_writer_unlock	Ολοκληρώνει εγγραφή, ξυπνάει αναγνώστες ή writers.

Β. Συνοπτικός Πίνακας με την κίνηση των Locks στο πρόγραμμα

Ενέργεια στο πρόγραμμα	Συνάρτηση που καλείται	Είδος κλειδώματος	Συνθήκη αναμονής	Πότε ξεκλειδώνει
db_add	<code>rwlock_writer_lock()</code>	Writer	Περιμένει να τελειώσουν οι αναγνώστες	<code>rwlock_writer_unlock()</code> μετά την εγγραφή
db_get	<code>rwlock_reader_lock()</code>	Reader	Περιμένει να ολοκληρωθεί ενεργή εγγραφή	<code>rwlock_reader_unlock()</code> μετά την ανάγνωση

Γ. Ακριβής Ροή Ενεργειών στο κλείδωμα

Περίπτωση Εγγραφής (db_add)

```

int db_add(DB* self, Variant* key, Variant* value)
{
    // Αποκλειστικό κλείδωμα εγγραφής πριν ξεκινήσει η προσθήκη στην βάση
    rwlock_writer_lock(&self->sst->rwlock);

    // Αν χρειάζεται compaction (συμπύκνωση), εκτελείται πριν την προσθήκη.
    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);
    }

    // Προσθήκη κλειδιού-τιμής στην memtable
    int ret = memtable_add(self->memtable, key, value);

    // Σεκλείδωμα εγγραφής, τώρα μπορούν να προχωρήσουν άλλοι readers/writers
    rwlock_writer_unlock(&self->sst->rwlock);

    return ret;
}

```

- Ο writer προσπαθεί να κλειδώσει (rwlock_writer_lock).
- Αναμένει να τελειώσουν όλοι οι τρέχοντες αναγνώστες.
- Όταν δεν υπάρχουν αναγνώστες (readers_count == 0), ξεκινάει την εγγραφή.
- Μετά την εγγραφή, ξεκλειδώνει (rwlock_writer_unlock) και ξυπνάει τα αναμένοντα νήματα (αναγνώστες ή writers).

Περίπτωση Ανάγνωσης (db_get)

```
int db_get(DB* self, Variant* key, Variant* value)
{
    int found = 0;

    // Παίρνουμε lock ανάγνωσης (πολλαπλοί αναγνώστες επιτρέπονται)
    rwlock_reader_lock(&self->sst->rwlock);

    // Πρώτα αναζητούμε στην memtable
    if (memtable_get(self->memtable->list, key, value)) {
        found = 1; // Βρέθηκε το κλειδί στην memtable
    } else {
        // Αν δεν βρέθηκε στην memtable, ψάχνουμε στο SST (Secondary Storage Table)
        found = sst_get(self->sst, key, value);
    }

    // Απελευθερώνουμε το lock ανάγνωσης
    rwlock_reader_unlock(&self->sst->rwlock);

    return found;
}
```

- Ο reader προσπαθεί να κλειδώσει (rwlock_reader_lock).
- Αν δεν υπάρχει ενεργός writer (writer_active == 0), μπορεί να διαβάσει αμέσως.
- Πολλοί αναγνώστες μπορούν να διαβάζουν ταυτόχρονα.
- Όταν τελειώσει η ανάγνωση, μειώνει το readers_count και ενημερώνει (rwlock_reader_unlock). Αν ήταν ο τελευταίος αναγνώστης, ενημερώνει τον writer που αναμένει.

Γ. Συνοπτικό Παράδειγμα Χρήσης

Σενάριο	Αναγνώστες	Γράφεις	Κατάσταση κλειδώματος
Αρχικά: κανένας ενεργός	✗	✗	Ελεύθερο
Ένας αναγνώστης ζητά πρόσβαση	✓	✗	Lock ανάγνωσης
Πολλοί αναγνώστες ζητούν πρόσβαση	✓ ✓ ✓ ...	✗	Lock πολλαπλών αναγνωστών
Ένας γράφων ζητά πρόσβαση	✓ ✓ ✓ ...	✗ ⏳	Writer περιμένει να τελειώσουν
Τελειώνουν οι αναγνώστες	✗	✓	Writer ξεκινά να γράφει
Γράφων ολοκληρώνει την εγγραφή	✗	✗	Ελεύθερο πάλι

5.3 Ιδέες Βελτίωσης Μηχανισμού

Η τρέχουσα χρήση ενός μοναδικού κλειδώματος (lock) για όλη τη δομή SST είναι λειτουργική αλλά όχι αποδοτική. Η πρόταση περιλαμβάνει το διαχωρισμό της αποθήκευσης σε μικρότερα τμήματα με ξεχωριστά locks, επιτρέποντας ταυτόχρονη πρόσβαση και μειώνοντας το contention μεταξύ threads. Η τεχνική RCU επιτρέπει στους αναγνώστες να λειτουργούν χωρίς σχεδόν καθόλου overhead locking, ενώ οι γραφείς δημιουργούν αντίγραφα, τα ενημερώνουν και τα αντικαθιστούν με atomic operations. Περαιτέρω παραλληλισμός μπορεί να επιτευχθεί

με sharding, διαχωρίζοντας τα δεδομένα σε ανεξάρτητα partitions, επιτρέποντας ταυτόχρονη πρόσβαση, μειώνοντας το contention σε global locks και αξιοποιώντας καλύτερα τα πολυπύρηνα συστήματα. Τέλος, η βελτιστοποίηση της διαδικασίας compaction μπορεί να γίνει σε background threads με συγχρονισμό μόνο του ελάχιστου απαραίτητου διαστήματος, αυξάνοντας σημαντικά την απόδοση ανάγνωσης/εγγραφής χωρίς μεγάλες καθυστερήσεις.

Συμπέρασμα

Ένα λοιπόν Ιδανικό πλάνος εξέλιξης θα ήταν:

Βήμα	Δράση	Δυσκολία	Απόδοση
1	Finer-grained locking	Μέτρια	Σημαντικά κέρδη
2	Background Async Compaction	Μέτρια	Σημαντικά κέρδη
3	Sharding/ Partitioning	Υψηλή	Μεγάλα κέρδη
4	Read-Copy-Update (RCU)	Υψηλή	Πολύ μεγάλα κέρδη

6. Πειραματική Διαδικασία

Όταν σχεδιάζουμε και υλοποιούμε έναν μηχανισμό συγχρονισμού – όπως αυτόν που περιγράφεται για την ταυτόχρονη πρόσβαση σε μια μηχανή βάσης δεδομένων – είναι κρίσιμο να επαληθεύουμε την απόδοση και τη σταθερότητά του σε πραγματικές ή προσομοιωμένες συνθήκες. Η **πειραματική διαδικασία** μάς επιτρέπει:

- Να **μετρήσουμε** πόσο αποτελεσματικός είναι ο μηχανισμός μας (χρόνοι εκτέλεσης, ρυθμοαπόδοση).
- Να **εντοπίσουμε αδυναμίες** (πιθανά σημεία συμφόρησης, αστάθειες, καθυστερήσεις).
- Να **συγκρίνουμε διαφορετικές παραμέτρους** (π.χ. πλήθος νημάτων, μέγεθος δεδομένων, διαφορετικά operations).

Προκειμένου να έχουμε **εγκυρότερα αποτελέσματα** και να μειώσουμε την επίδραση της τυχαιότητας των συστηματικών διεργασιών, **εκτελέσαμε κάθε πείραμα 5 φορές** και **κρατήσαμε τον μέσο όρο των μετρήσεων**. Με αυτόν τον τρόπο, επιτυγχάνουμε στατιστικά πιο έγκυρη εκτίμηση του χρόνου και της απόδοσης.

6.1 Παράμετροι των Πειραμάτων και τρόπος μελέτης

Τα πειράματα διενεργούνται για τρία διαφορετικά **operation modes**:

1. **write**
2. **read**
3. **readwrite** (με διάφορα ποσοστά αναγνώσεων)

Για την **περίπτωση readwrite**, χρησιμοποιούμε επιπρόσθετα τις τιμές στα **readwrite percentages = [10, 30, 50, 70, 90]**, ώστε να καλύψουμε σενάρια με κυρίως αναγνώσεις ή κυρίως εγγραφές.

Ο αριθμός των νημάτων (**threads**) λαμβάνει τιμές από **[1, 2, 4, 8, 16, 32, 64, 100]**, και ο αριθμός των διεργασιών (π.χ., κλειδιών που θα διαβαστούν/γραφτούν), δηλαδή τα **counts**, είναι **[10000, 50000, 100000, 250000, 500000, 750000, 1000000]**.

Η εκτέλεση όλων των πειραμάτων γίνεται αυτόματα από ένα script που δεν θα αναφέρουμε εδώ για την αυτοματοποίηση της διαδικασίας

Παρότι καταγράφουμε επίσης τον μέσο χρόνο ανά thread (secPerThread), δεν θα αποτελέσει αντικείμενο λεπτομερούς μελέτης. Εστιάζουμε κυρίως στην **απόδοση (ρυθμαπόδοση) - (ops/sec)**, το **συνολικό χρόνο (totalTime)**, και για ποσοστά **readwrite 30-70, 50-50, 70-30**, καθώς δίνουν σαφέστερη εικόνα για το πώς κλιμακώνεται η απόδοση του μηχανισμού όταν αυξάνονται τα threads ή τα counts.

Συνάρτηση Αποθήκευσης αποτελεσμάτων

```
// Συνάρτηση που αποθηκεύει τα αποτελέσματα από κάθε benchmark σε ένα αρχείο benchmark_results.txt
void saveResultsToFile(BenchResults *res, BasicBenchArgs *basicArgs, AdvancedBenchArgs *advArgs, Datas *datas, int opIdx) {
    FILE* file = fopen("/home/myy601/kiwi/kiwi-source/bench/benchmark_results.txt", "a"); // append mode

    if(opIdx == 0){
        fprintf(file, "\n===== Ορισμότα benchmark =====\n");
        fprintf(file, "Λειτουργία: %s\n", basicArgs->operationMode);
        fprintf(file, "Συνολικές εργασίες: %ld\n", basicArgs->operationCount);
        fprintf(file, "Αριθμός νημάτων: %ld\n", advArgs->threadCount);
        fprintf(file, "Τυχαία πρόβαση: %s\n", advArgs->isRandom ? "Ναι" : "Όχι");
        if (strcmp(basicArgs->operationMode, "readwrite") == 0) {
            fprintf(file, "Ποσοστό αναγνώσεων: %.2f%\n", advArgs->readPercentage);
        }
        fprintf(file, "Συνολικός χρόνος Benchmark: %.4Lf sec\n", datas->GlobalTimer.totalTime);
    }

    fprintf(file, "\n===== Αποτελέσματα benchmark %s =====\n", res->threadArgs[0].isReader ? "READ" : "WRITE");
    fprintf(file, "Threads που χρησιμοποιήθηκαν: %ld\n", res->totalThreads);
    fprintf(file, "Συνολικός αριθμός εργασιών: %ld\n", res->totalOperations);
    fprintf(file, "Συνολικός χρόνος operation: %.4Lf sec\n", res->OperationTimer.totalTime);
    fprintf(file, "Συνολική απόδοση (ops/sec): %.2Lf\n", res->opsPerSecond);
    fprintf(file, "Μέσος χρόνος ανά εργασία (sec/op): %.6Lf sec\n", res->secPerOp);
    fprintf(file, "Μέσος χρόνος ανά thread (sec/thread): %.6Lf sec\n", res->secPerThread);

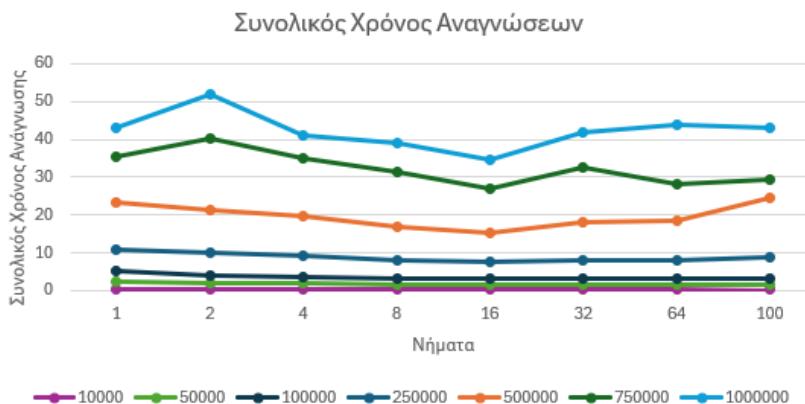
    if (res->threadArgs[0].isReader) {
        fprintf(file, "Συνολικό κλειδιά που βρέθηκαν: %ld\n", res->keysRetrieved);
    }

    fclose(file);
}
```

6.2 Σχολιασμός Πειραμάτων Ανάγνωση

6.2.1 Συνολικός Χρόνος Εκτέλεσης (Read)

		Operation Counts for Read mode							
		Συνολικός χρόνος	10000	50000	100000	250000	500000	750000	1000000
t h r e a d s	1	1	0,5428	2,1797	5,2199	10,841	23,192	35,597	43,044
	2	2	0,522	1,9957	3,9079	10,089	21,418	40,084	51,826
	4	4	0,3833	1,7945	3,5413	9,0742	19,866	35,202	41,208
	8	8	0,4443	1,6412	3,2498	8,0897	16,817	31,565	38,948
	16	16	0,398	1,5991	3,121	7,6634	15,428	27,147	34,582
	32	32	0,3774	1,6743	3,0241	8,1753	17,916	32,753	41,719
	64	64	0,3983	1,7452	3,3268	7,9824	18,596	28,252	43,788
	100	100	0,3533	1,4636	3,0594	8,9031	24,643	29,311	43,186



1. Με λίγα threads (π.χ. 1 ή 2): Η εκτέλεση έχει σχετικά μικρό βαθμό παραλληλισμού. Ωστόσο, το memtable και τα SST files προσπελαύνονται δίχως ανταγωνισμό από άλλους αναγνώστες. Ο συνολικός χρόνος μεγαλώνει **ανάλογα με τον αριθμό των operations** (π.χ. στα 250.000 ή 500.000 αναγνώσεις), αλλά σε μικρά μεγέθη ενδέχεται να είναι αρκετά γρήγορος. Στον πίνακα βλέπουμε ότι όταν έχουμε μόνο 1 thread, ο χρόνος αυξάνει σταδιακά καθώς αυξάνουν τα operations, κάτι αναμενόμενο (π.χ. 0.5 δευτερόλεπτα για 10.000, 2+ δευτερόλεπτα για 50.000 κ.λπ.).

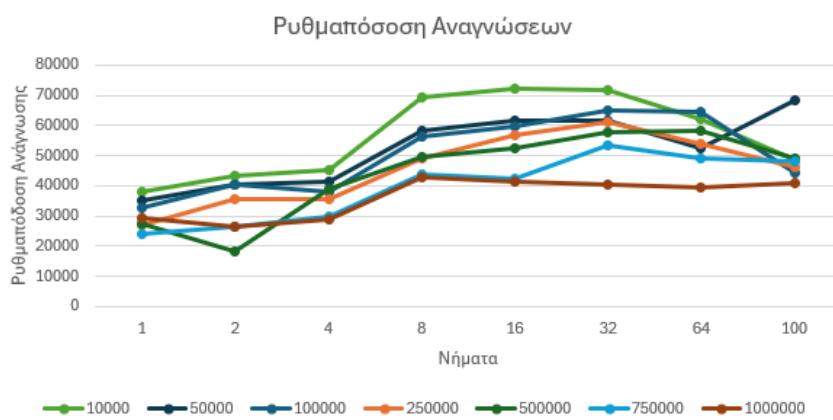
2. Με περισσότερα threads (4, 8, 16, 32...):

Οι αναγνώστες μπορούν να «μπαίνουν» ταυτόχρονα στη βάση, οπότε συχνά βλέπουμε ότι ο συνολικός χρόνος μειώνεται ή παραμένει σε λογικά επίπεδα, παρά το ότι οι operation counts αυξάνουν. Δηλαδή, καθώς πολλαπλασιάζονται οι αναγνώστες, εκτελούνται παράλληλα πολλές αναζητήσεις.

- Π.χ. στα 16 threads, αν ο πίνακας δείχνει ότι ο χρόνος έχει πέσει σημαντικά σε σχέση με τα 2 threads, σημαίνει ότι κερδίζουμε από τον παραλληλισμό (μέχρι βέβαια να «κορεστεί» ο επεξεργαστής ή η μνήμη).
- Σε ακόμη μεγαλύτερα νούμερα threads (π.χ. 64, 100), συχνά βλέπουμε ένα **σημείο κορεσμού**: ο χρόνος δεν μειώνεται άλλο ή ενίοτε αυξάνει λίγο, λόγω overhead συγχρονισμού ή φόρτου CPU/memory.

6.2.2 Ρυθμαπόδοση (Throughput) - ops/sec (Read)

	Ρυθμαπόδοση	Operation Counts for Read mode						
		10000	50000	100000	250000	500000	750000	1000000
t h r e a d s	1	18430	22940	19158	23062	21559	21069	23232
	2	19213	25101	25604	24784	23353	18729	19302
	4	27527	28097	28313	27611	25179	21312	24281
	8	27036	30757	30952	30977	29801	23779	25690
	16	27438	31999	32719	32831	32463	27658	28999
	32	28865	30511	33304	30731	28018	22939	24000
	64	27433	29227	30348	31924	26937	26595	22870
	100	30686	35018	33036	28201	20491	25718	23289



1. Με 1 thread: Συνήθως έχει μικρό throughput γιατί δεν υπάρχει παράλληλη εκτέλεση. Για λίγες χιλιάδες operations, το throughput μπορεί να φαίνεται μεγάλο, αλλά αυξάνοντας πολύ τα operations βλέπουμε ότι ο 1 πυρήνας/νήμα δεν αρκεί.

2. Με 2, 4, 8 threads: Παρατηρείται **αύξηση** του throughput, επειδή πολλοί αναγνώστες μπορούν να εκτελούν ταυτόχρονα τις αναζητήσεις (memtable ή sst_get). Το reader-writer lock δεν εμποδίζει τους αναγνώστες μεταξύ τους· τους επιτρέπει να μπουν όλοι μαζί. Άρα, όσο δεν υπάρχει συγχρονισμός με writer, κερδίζουμε σε παράλληλη επεξεργασία.

3. Με 16, 32, 64+ threads: Πολλές φορές βλέπουμε ακόμα **περαιτέρω αύξηση** της ρυθμαπόδοσης, μέχρι να φτάσουμε στα όρια του συστήματος (CPU, μνήμη, cache). Όταν τα όρια ξεπεραστούν, το throughput μπορεί να σταθεροποιηθεί ή και να πέσει, λόγω overhead, context switching, και πιθανής συμφόρησης στον δίαυλο μνήμης (memory bus).

Συνολική Εικόνα:

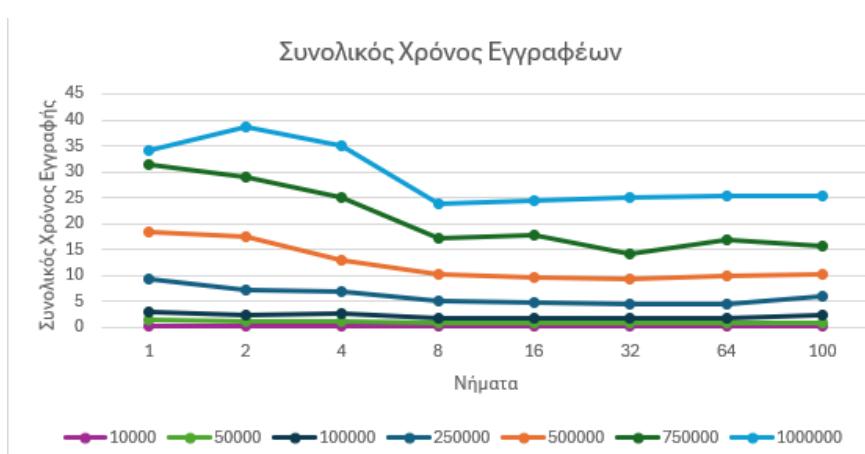
Η αύξηση του throughput συνήθως **δεν** είναι γραμμική όσο μεγαλώνει ο αριθμός των threads. Υπάρχει **φαινόμενο φθίνουσας απόδοσης** (diminishing returns) από κάποια κλίμακα και πάνω. Επίσης, ο μηχανισμός κλειδώματος για τους αναγνώστες εδώ έχει μικρό κόστος, διότι μπαίνουν όλοι μαζί (shared lock). Σημαντικό ρόλο παίζει και το πόση επεξεργαστική ισχύ (ή μνήμη) υπάρχει για να εξυπηρετηθούν οι αναγνώσεις. Εν ολίγοις, τα πειράματα επιβεβαιώνουν ότι η συγκεκριμένη κλειδαριά reader-writer, σε σενάριο μόνο-ανάγνωσης (Read mode), επιτρέπει ικανοποιητική κλιμάκωση του throughput με την αύξηση των threads, μέχρι να εμφανιστούν φυσικά όρια του συστήματος (CPU cores, memory bandwidth). Αυτό αποτυπώνεται τόσο στη **μείωση του συνολικού χρόνου** όσο και στην **αύξηση της ρυθμαπόδοσης**, τουλάχιστον μέχρι έναν βαθμό.

6.3 Σχολιασμός Πειραμάτων Εγγραφέων

Στο πλαίσιο της πολυνηματικής υλοποίησης, η πρόκληση για τις εγγραφές είναι να διασφαλιστεί ότι πολλαπλά νήματα μπορούν να προσθέτουν δεδομένα ταυτόχρονα, χωρίς να δημιουργούνται συνθήκες ανταγωνισμού με το νήμα συγχώνευσης, και χωρίς να επηρεάζεται αρνητικά η απόδοση λόγω υπερβολικού συγχρονισμού.

6.3.1 Συνολικός Χρόνος Εκτέλεσης (Write)

		Operation Counts for Write mode						
Συνολικός χρόνος		10000	50000	100000	250000	500000	750000	1000000
t	1	0,2639	1,4288	3,077	9,2734	18,281	31,292	34,121
h	2	0,2387	1,2321	2,5084	7,1844	17,464	29,017	38,68
r	4	0,2477	1,2474	2,7421	7,0581	12,894	25,048	34,962
e	8	0,1666	0,9523	1,8337	5,1143	10,298	17,145	23,864
a	16	0,1556	0,855	1,7567	4,7021	9,7887	17,76	24,494
d	32	0,1643	0,8238	1,6715	4,5479	9,2681	14,321	24,928
s	64	0,1927	0,9827	1,7823	4,6488	10,067	17,013	25,445
	100	0,2307	0,9769	2,4799	5,9818	10,255	15,753	25,408



1. Επίδραση αποκλειστικού κλειδώματος για Writer

- Σε αντίθεση με τους αναγνώστες, ο writer lock είναι **αποκλειστικός**. Αυτό σημαίνει ότι όταν ένα thread γράφει, κανένας άλλος (reader ή writer) δεν

μπορεί να έχει πρόσβαση στη δομή. Έτσι, αν έχουμε πολλούς writers, μπαίνουν σε σειρά αναμονής (serialization).

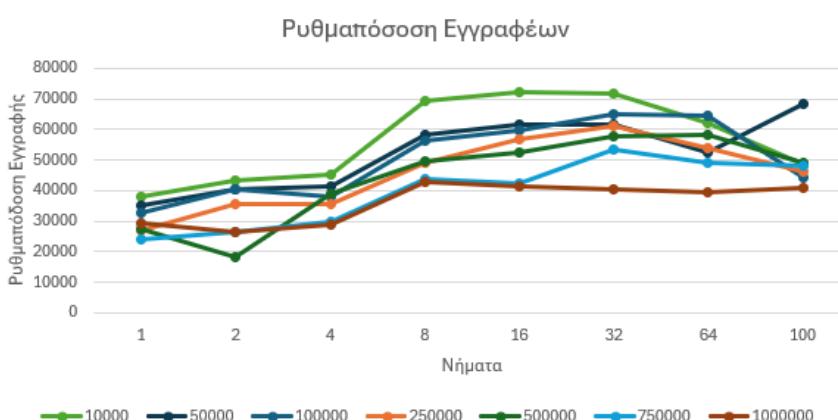
- Στους πίνακες, φαίνεται ότι καθώς αυξάνονται τα threads (π.χ. 1, 2, 4, 8...), ο συνολικός χρόνος δεν πέφτει τόσο δραματικά όσο στους readers. Σε πολλά νήματα, βλέπουμε **ορισμένες περιπτώσεις** όπου ο χρόνος ακόμα και **αυξάνεται**, επειδή κάθε writer περιμένει τους άλλους να τελειώσουν, με αποτέλεσμα περισσότερη αναμονή και context switching.

2. Compaction / Συγχώνευση Memtable

- Στη λειτουργία Write, όταν εισάγονται πάρα πολλά κλειδιά, η memtable μπορεί να χρειάζεται συχνότερα compaction (συγχώνευση στον δίσκο). Αυτό συνεπάγεται επιπλέον αποκλειστικό κλείδωμα, διότι το compaction γράφει επίσης στο SST. Έτσι, όσο αυξάνεται ο αριθμός εργασιών (π.χ. 10000 → 100000 → 500000 κ.λπ.), οι compactions μπορεί να γίνονται συχνότερες και να αυξάνουν σημαντικά τον συνολικό χρόνο.

6.3.2 Ρυθμαπόδοση (Throughput) - ops/sec (Write)

Operation Counts for Write mode								
	10000	50000	100000	250000	500000	750000	1000000	
t	1 ➔ 37917 ↘	34996 ↘	32500 ↘	26960 ↘	27351 ↘	23969 ↘	29308	
h	2 ➔ 43353 ➡	40667 ➡	40307 ↘	35402 ↘	18207 ↘	26439 ↘	26291	
r	4 ➔ 45461 ➡	41362 ➡	37921 ↘	35721 ➡	38793 ↘	29952 ↘	28682	
e	8 ➗ 69545 ➗	58188 ➗	56507 ➡	48967 ➡	49732 ➡	44040 ➡	43009	
a	16 ➗ 72441 ➗	61425 ➗	59911 ➗	57038 ➡	52645 ➡	42463 ➡	41491	
d	32 ➗ 71856 ➗	61720 ➗	64892 ➗	61287 ➗	57824 ➡	53283 ➡	40295	
s	64 ➗ 62370 ➡	52330 ➗	64473 ➡	54122 ➗	58428 ➡	49329 ➡	39416	
	100 ➡ 48491 ➗	68169 ➡	44126 ➡	45998 ➡	49138 ➡	48223 ➗	41037	



1. Περιορισμένη κλιμάκωση

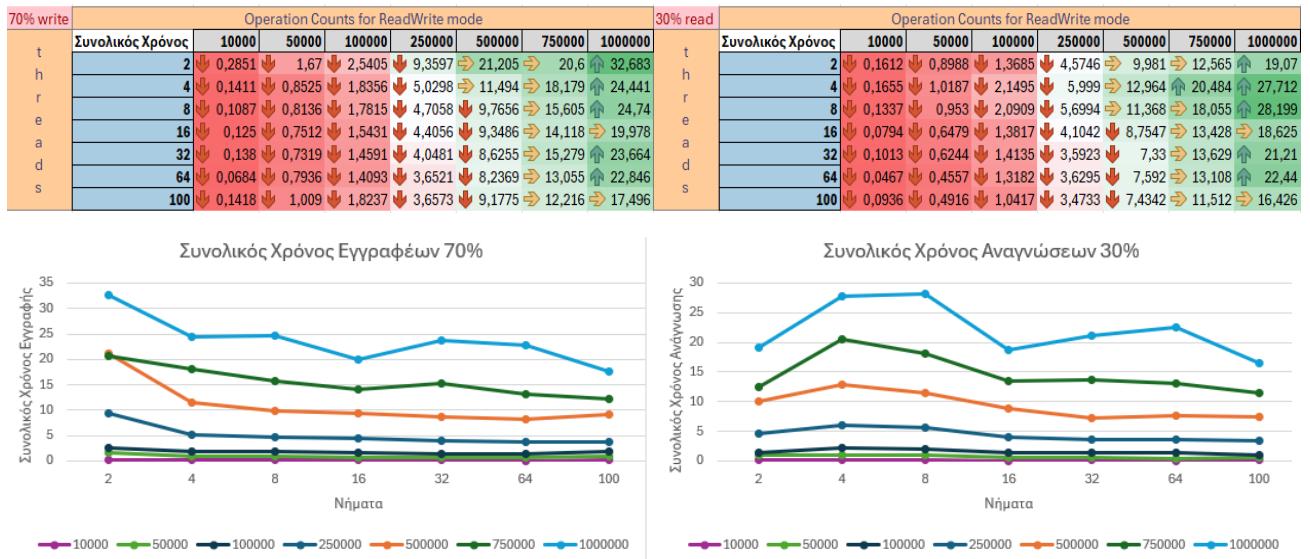
- Όπως προαναφέρθηκε, επειδή μόνο **ένας** writer μπορεί να δουλέψει κάθε φορά, η αύξηση του αριθμού των νημάτων δεν μεταφράζεται πάντα σε ανάλογη αύξηση ρυθμαπόδοσης. Η συγχρονιστική συμφόρηση (writer lock) μειώνει το όφελος.
- Παρότι μπορεί να δούμε κάποια βελτίωση από 1 thread σε 2-3 threads (π.χ. εκμετάλλευση CPU σε idle time, κ.λπ.), σε πολύ μεγάλο αριθμό threads (16, 32, 64...), η ρυθμαπόδοση συχνά **πέφτει** (λόγω μεγαλύτερης ουράς αναμονής).

2. Υψηλά operation counts

- Όσο περισσότερα inserts/write requests γίνονται, τόσο πιο συχνά θα ενεργοποιούνται επιπλέον διαδικασίες, όπως compaction, flush της memtable στον δίσκο κ.ά. Αυτές οι διαδικασίες αναλαμβάνουν επίσης αποκλειστικό κλείδωμα, μειώνοντας τη ρυθμαπόδοση σε περιόδους όπου πολλές εγγραφές συσσωρεύονται.

6.4 Σχολιασμός Πειραμάτων Μεικτής Λειτουργίας (ReadWrite)

6.4.1 Συνολικός Χρόνος (Total Time) για 70% Write / 30% Read



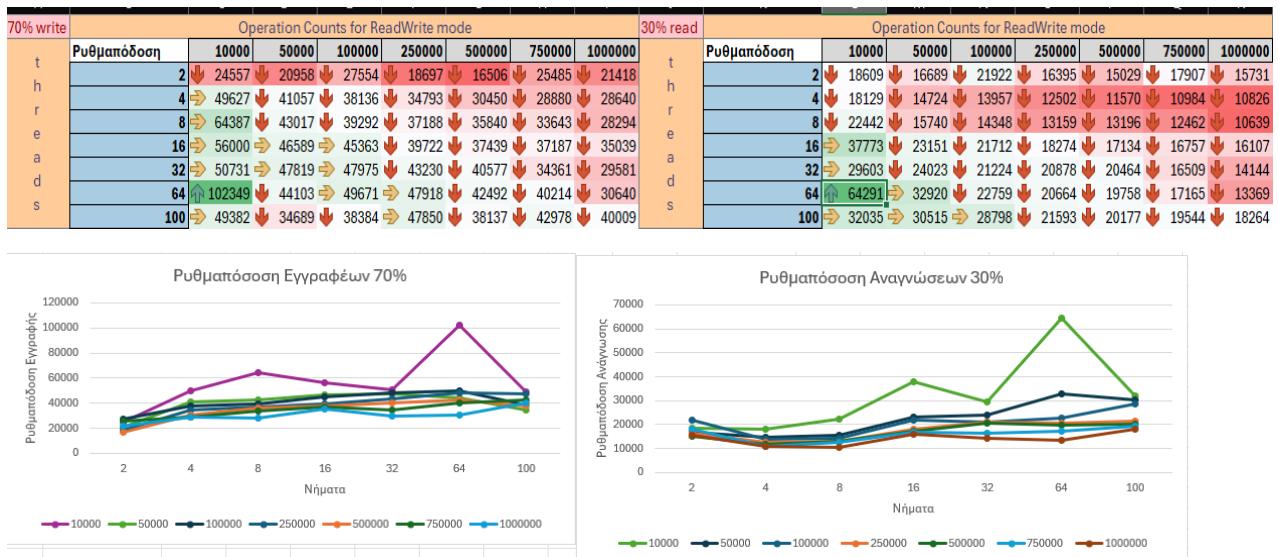
1. Κυριαρχία του Writer Lock

- Επειδή οι περισσότερες εργασίες (70%) αφορούν **εγγραφές**, ο μηχανισμός reader-writer lock αναγκάζεται να μπαίνει πολύ συχνά σε **αποκλειστική** λειτουργία (writer lock). Αυτό σημαίνει ότι κάθε φορά που κάποιος writer εκτελείται, οι υπόλοιποι writers πρέπει να περιμένουν σε ουρά και **κανένας** αναγνώστης δεν μπορεί να αποκτήσει πρόσβαση παράλληλα.
- Το γεγονός ότι έχουμε και 30% reads δεν αρκεί για να εκμεταλλευτεί στο έπακρο το parallel read, γιατί τα writers έχουν πολύ πιο συχνά προτεραιότητα να κλειδώσουν αποκλειστικά. Έτσι, οι αναγνώστες σπάνια «μπαίνουν» ταυτόχρονα σε μεγάλες ομάδες, καθώς διακόπτονται από τα συνεχή writes.

2. Αύξηση αριθμού νημάτων

- Όταν αυξάνουμε τα threads (π.χ. $1 \rightarrow 4 \rightarrow 8 \rightarrow 16$ κ.λπ.), αρχικά μπορεί να δούμε **ελαφρά μείωση** του συνολικού χρόνου (τα reads τρέχουν λίγο πιο παράλληλα), όμως πέρα από ένα όριο οι πολλοί writers ανταγωνίζονται μεταξύ τους σε σειριακή ουρά. Αυτό συνήθως οδηγεί σε **αύξηση** του χρόνου (ή τουλάχιστον όχι σε γραμμική μείωση), γιατί κάθε writer περιμένει περισσότερο.
- Οι αναγνώστες μπορεί να μπαίνουν μαζί, αλλά το κέρδος τους είναι περιορισμένο: 30% είναι αναγνώσεις, 70% είναι οι αποκλειστικές εγγραφές. Όσο πιο πολλά threads-writers υπάρχουν, τόσο πιο μεγάλη «αναμονή» και πιο έντονος ο συγχρονισμός.

6.4.2 Ρυθμαπόδιση (ops/sec) για 70% Write / 30% Read



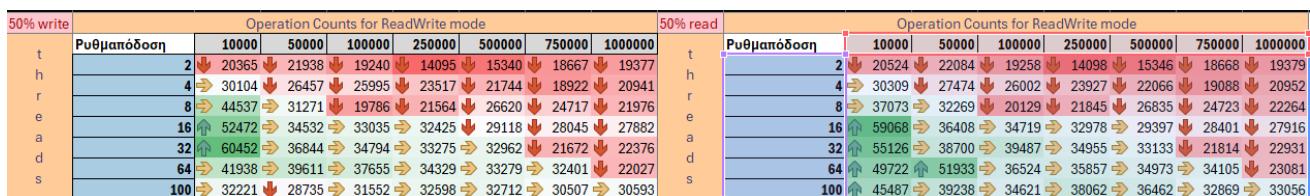
1. Περιορισμένη κλιμάκωση

- Παρόλο που το 30% των αναγνώσεων θα μπορούσαν να εκτελούνται παράλληλα (και πράγματι κερδίζουμε κάποια βελτίωση από την πολυνηματικότητα), το 70% εγγραφών θέτει σειριακό φραγμό. Στην πράξη, οι writers περιμένουν ο ένας τον άλλο και αυτό περιορίζει δραστικά το πόσο «ψηφλά» μπορεί να ανέβει η συνολική ρυθμαπόδιση.
- Συχνά βλέπουμε ότι από 1 νήμα σε 2-4 νήματα υπάρχει κάποια αύξηση στις ops/sec, επειδή τα reads επωφελούνται, αλλά πέρα από έναν αριθμό (π.χ. 8 ή 16 threads) η ρυθμαπόδιση δεν ανεβαίνει με τον ίδιο ρυθμό. Μπορεί ακόμα και να πέφτει ελαφρώς λόγω overhead σε scheduling και συνεχείς εναλλαγές των writers.

2. Επίδραση 30% Read

- Το γεγονός ότι μόνο 30% είναι reads σημαίνει πως η βελτίωση από το parallel read δεν αρκεί για να σηκώσει δραματικά την ολική ρυθμαπόδιση. Η πλειοψηφία των εργασιών (70%) είναι writes και γίνονται σε αποκλειστική πρόσβαση.
- Παρ' όλα αυτά, το 30% των αναγνώσεων δίνει ένα κάποιο κέρδος, σε σχέση με το σενάριο «100% write», καθώς τουλάχιστον εκείνα τα 30% μπορούν να τρέχουν ταυτόχρονα. Αυτό φαίνεται σε ορισμένες τιμές threads, όπου η ρυθμαπόδιση είναι ελαφρώς καλύτερη από αντίστοιχο σενάριο 100% write.

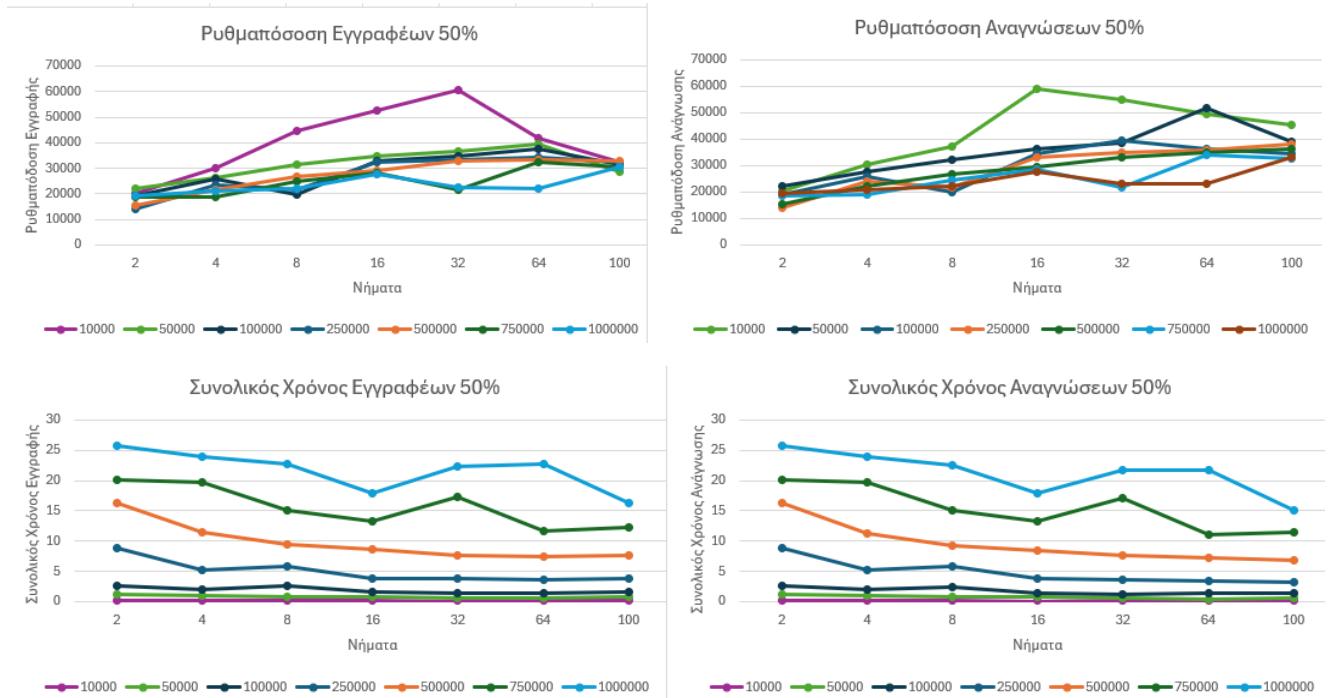
6.4.3 Συνολικός Χρόνος και Ρυθμαπόδιση (Total Time) για 50% Write / 50% Read



t h r e a d s	Operation Counts for ReadWrite mode							t h r e a d s	Operation Counts for ReadWrite mode						
	Συνολικός Χρόνος	10000	50000	100000	250000	500000	750000	1000000	Συνολικός Χρόνος	10000	50000	100000	250000	500000	750000
2	0,2455	1,1396	2,5988	8,8684	16,297	20,089	25,804	2	0,2436	1,132	2,5963	8,8664	16,291	20,088	25,801
4	0,1661	0,9449	1,9234	5,3153	11,497	19,819	23,876	4	0,165	0,9099	1,9229	5,2243	11,33	19,646	23,864
8	0,1123	0,7995	2,527	5,7968	9,3913	15,172	22,752	8	0,1349	0,7747	2,484	5,7222	9,3161	15,168	22,458
16	0,0953	0,724	1,5136	3,8551	8,5859	13,371	17,933	16	0,0846	0,6867	1,4401	3,7904	8,5043	13,204	17,911
32	0,0827	0,6785	1,437	3,7566	7,5844	17,303	22,346	32	0,0907	0,646	1,2662	3,576	7,5454	17,19	21,804
64	0,1192	0,6311	1,3278	3,6412	7,5123	11,574	22,699	64	0,1006	0,4814	1,369	3,486	7,1484	10,995	21,663
100	0,1552	0,87	1,5847	3,8346	7,6423	12,292	16,344	100	0,1099	0,6371	1,4442	3,2841	6,8564	11,409	15,126

1. Ισορροπία ανάμεσα σε Parallel Read και Exclusive Write

- Σε αντίθεση με τα ακραία σενάρια (70% ή 90% εγγραφές), εδώ έχουμε μια πιο **ισορροπημένη** κατανομή μεταξύ αναγνωστών και εγγραφέων. Αυτό σημαίνει ότι οι αναγνώστες μπορούν όντως να τρέχουν παράλληλα (όταν δεν εκτελείται κάποιος writer), ενώ οι writers, αν και αποκλειστικοί, δεν αποτελούν την κυριαρχη πλειονότητα.
- Το αποτέλεσμα είναι ότι **μερικές φορές** βλέπουμε καλύτερη κλιμάκωση (throughput) από τις περιπτώσεις όπου οι εγγραφές ήταν 70%+ η παράλληλη εκτέλεση των 50% αναγνώσεων δίνει σημαντικό όφελος. Από την άλλη, εξακολουθεί να υπάρχει ο περιορισμός του writer lock: όταν πολλά threads γράφουν, περιμένουν το ένα το άλλο.



2. Συγχρονισμός Reader-Writer

- Με 50% read, οι πιθανότητες να υπάρχουν ταυτόχρονα πολλοί αναγνώστες είναι αρκετά μεγάλες. Αυτό **βελτιώνει** τη συνολική απόδοση (ops/sec) σε σχέση με ένα σενάριο 100% write, γιατί οι αναγνώσεις δεν χρειάζονται αποκλειστική πρόσβαση και μπορούν να εκτελούνται παράλληλα.
- Παράλληλα, επειδή οι εγγραφές καλύπτουν επίσης το 50% των εργασιών, το σύστημα εισάγει έναν «φραγμό» κάθε φορά που γράφουν πολλοί writers. Έτσι, όταν αυξάνονται τα threads, δεν έχουμε «άπειρη» βελτίωση. Μεγάλος αριθμός νημάτων σημαίνει πολλούς writers σε ουρά, οπότε μετά από κάποιο σημείο οι συγκρούσεις αυξάνονται.

2. Επίδραση του Αριθμού Εργασιών (Operation Counts)

1. Μικρά Operation Counts

- Για μικρά πλήθη (π.χ. 10.000 εργασίες), ο γενικός χρόνος είναι αρκετά χαμηλός. Εδώ, η διαφορά ανάμεσα σε 1 ή 2 threads δεν φαίνεται τεράστια, διότι το overhead του scheduling και των locks μπορεί να «κρύψει» τα οφέλη της παράλληλης ανάγνωσης.
- Επίσης, όταν τα operation counts είναι μικρά, η memtable μπορεί να μη φτάσει συχνά στα όρια compaction, άρα έχουμε λιγότερες «υποχρεωτικές» καθυστερήσεις.

2. Μεγάλα Operation Counts

- Όταν το πλήθος εργασιών ανεβαίνει (π.χ. 250.000, 500.000 ή 1.000.000), αρχίζουμε να βλέπουμε πιο **σταθερή** τάση. Τα reads κερδίζουν από την παράλληλη εκτέλεση (ειδικά αν υπάρχουν 4, 8 ή περισσότερα threads). Όμως, η παρουσία 50% εγγραφών σημαίνει ότι, κάθε φορά που ένας writer εκτελείται, μπλοκάρει τους υπόλοιπους writers **και** τους readers.
- Για μεσαία ή μεγάλα operation counts, τα φαινόμενα compaction είναι επίσης πιο συχνά. Καθώς η memtable γεμίζει, ενεργοποιείται συγχώνευση (compaction), η οποία και αυτή χρειάζεται αποκλειστικό writer lock. Αυτό συχνά αυξάνει τον συνολικό χρόνο ή συγκρατεί τη ρυθμαπόδοση κάτω από μια γραμμική κλιμάκωση.

3. Επίδραση του Αριθμού Νημάτων (Threads)

1. Αρχική Αύξηση Threads

- Όταν ανεβάζουμε από 1 σε 2, 4, 8 threads, παρατηρούμε συνήθως **σημαντική βελτίωση** στην παράλληλη εξυπηρέτηση των αναγνώσεων, άρα η ρυθμαπόδοση (ops/sec) ανεβαίνει και ο συνολικός χρόνος πέφτει. Αυτό είναι εμφανές ιδίως σε μέτρια operation counts, όπου τα reads εκμεταλλεύονται το parallel read lock.
- Ωστόσο, η ταυτόχρονη παρουσία 50% writers σημαίνει πως δεν μπορούμε να έχουμε «κινόν» αναγνώστες ενεργούς. Όταν ένας writer ζητάει πρόσβαση, όλοι οι αναγνώστες μπλοκάρονται και ο writer εκτελείται αποκλειστικά.

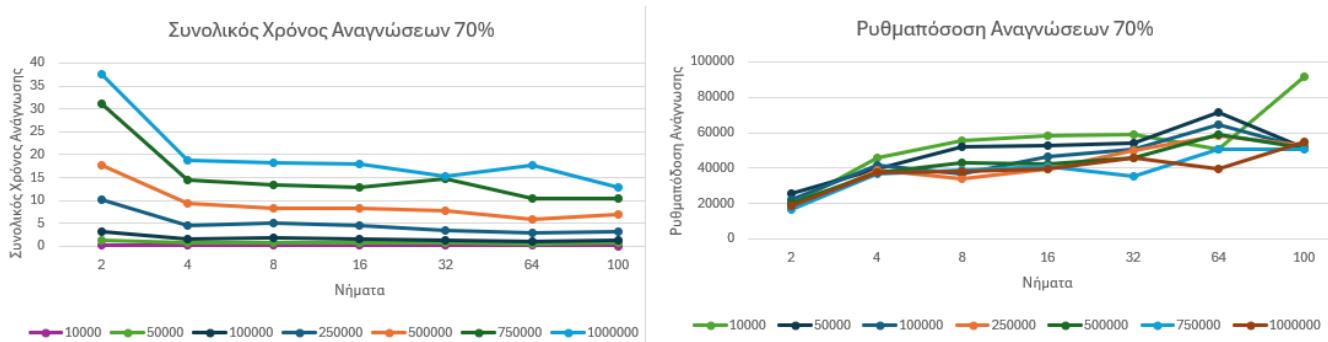
2. Πολλοί Writers σε Ύψηλα Threads

- Όταν τα threads γίνονται πολύ πολλά (16, 32, 64 ή 100), ενώ τα reads μπορούν να επωφεληθούν, οι πολλοί writers **δυσχεραίνουν** την κλιμάκωση. Για κάθε write, τα υπόλοιπα write-threads μπαίνουν σε ουρά, και το σύστημα κάνει συχνές ενολλαγές (context switches).
- Έτσι, η καμπύλη βελτίωσης «ισιώνει» ή ακόμα μπορεί να παρατηρηθεί **αύξηση** χρόνου για πολύ μεγάλα thread counts, λόγω του overhead του λειτουργικού συστήματος στο scheduling και των συχνών lock/unlock.

6.4.4 Συνολικός Χρόνος και Ρυθμαπόδοση (Total Time) για 30% Write / 70% Read

Σχολιασμός της Ρυθμαπόδοσης (Throughput)

30% write	Operation Counts for ReadWrite mode							70% read	Operation Counts for ReadWrite mode							
	Ρυθμαπόδοση	10000	50000	100000	250000	500000	750000	1000000	Ρυθμαπόδοση	10000	50000	100000	250000	500000	750000	1000000
t h r e a d s	2	17423	25667	23306	15841	18742	14702	19355	2	19539	25982	22491	17264	19821	16905	18593
	4	17965	15100	16051	15243	14356	14028	14602	4	45636	39443	41565	38826	37570	36459	37149
	8	20558	19881	14299	13384	16471	15225	14531	8	55652	51804	36643	34153	42845	39099	38145
	16	27995	24660	21411	19191	18573	18648	18294	16	58182	52631	46747	39429	42113	40700	39190
	32	46201	23220	23656	23104	20087	15524	20726	32	58784	54062	50619	49815	45553	35672	45903
	64	26290	24468	24243	23935	22727	20973	16327	64	50359	71420	64692	58334	58712	50524	39487
	100	31809	17294	21930	22009	21108	21056	20759	100	91719	51597	51239	52809	51191	50431	54706



Για τις αναγνώσεις (70%):

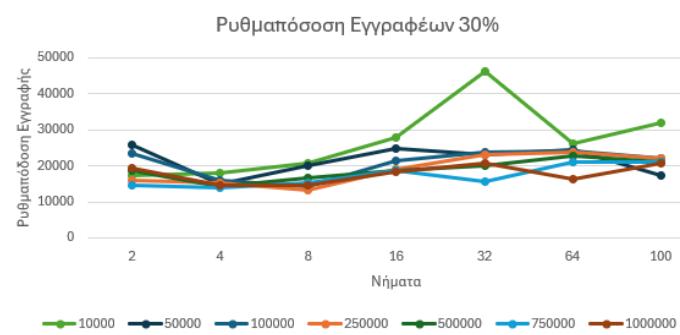
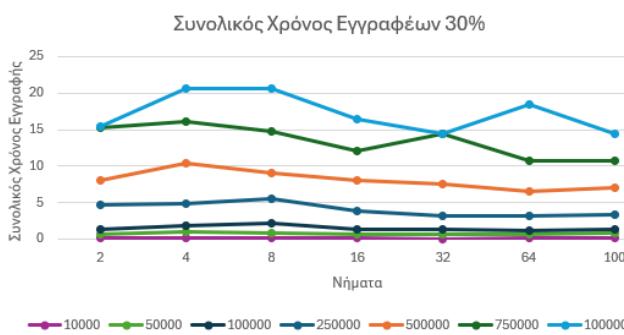
- Παρατηρείται ότι για σχετικά χαμηλά workloads (10.000–100.000 operations), η απόδοση των αναγνώσεων είναι υψηλή και αυξάνεται με τον αριθμό των threads.
- Όσο αυξάνεται ο αριθμός των operations (>250.000), η ρυθμαπόδοση αρχικά σταθεροποιείται και κατόπιν μειώνεται σταδιακά, κυρίως στα threads μέχρι και τα 32.
- Στα 64 και 100 threads εμφανίζεται μια έντονη αύξηση της απόδοσης για χαμηλά operation counts (π.χ., 71.420 στα 64 threads και 91.719 στα 100 threads για 10.000 operations).
- Η αιτία αυτής της υψηλής απόδοσης με πολλά threads και λίγα operations σχετίζεται με τον τρόπο που λειτουργεί η κλειδαριά (reader-writer lock). Ειδικότερα, πολλοί ταυτόχρονοι αναγνώστες επιτρέπονται, και όταν η αναλογία αναγνωστών είναι πολύ μεγάλη σε σχέση με τους εγγραφείς (όπως εδώ 70%), οι αναγνώστες έχουν λιγότερη αναμονή. Αυτό οδηγεί σε σημαντική αύξηση του throughput για μικρά μεγέθη δεδομένων και μεγάλο αριθμό threads.

Για τις εγγραφές (30%):

- Η απόδοση των εγγραφών εμφανίζει συνολικά χαμηλότερη τιμή σε σχέση με τις αναγνώσεις, κάτι που είναι φυσιολογικό λόγω της αποκλειστικής πρόσβασης που απαιτούν οι εγγραφές.
- Η μείωση του throughput όσο αυξάνονται οι εργασίες είναι αναμενόμενη και οφείλεται στην αύξηση των compactions και γενικότερα στην καθυστέρηση από τον αποκλειστικό αποκλεισμό (mutex lock) των εγγραφών.
- Παρατηρείται ότι όσο αυξάνονται τα threads (>32 threads), οι εγγραφές πλήττονται περισσότερο, διότι αυξάνεται το contention (ανταγωνισμός) μεταξύ εγγραφέων και αναγνωστών, και συνεπώς μειώνεται η συνολική απόδοση.

Σχολιασμός του Συνολικού Χρόνου Εκτέλεσης

Συνολικός Χρόνος	Operation Counts for ReadWrite mode							Συνολικός Χρόνος	Operation Counts for ReadWrite mode							
	10000	50000	100000	250000	500000	750000	1000000		10000	50000	100000	250000	500000	750000	1000000	
t h r e a d s	2	0,1722	0,5844	1,2872	4,7346	8,0034	15,304	15,5	2	0,3583	1,3471	3,1124	10,137	17,658	31,056	37,646
	4	0,167	0,9934	1,869	4,9204	10,449	16,039	20,545	4	0,1534	0,8874	1,6841	4,5073	9,316	14,4	18,843
	8	0,1459	0,7545	2,098	5,6038	9,1067	14,778	20,646	8	0,1258	0,6756	1,9103	5,1241	8,1689	13,428	18,351
	16	0,1072	0,6083	1,4012	3,9082	8,0762	12,066	16,399	16	0,1204	0,665	1,4974	4,4383	8,3109	12,899	17,862
	32	0,0649	0,646	1,2682	3,2462	7,4674	14,494	14,475	32	0,1191	0,6474	1,3629	3,513	7,6834	14,717	15,25
	64	0,1141	0,613	1,2375	3,1335	6,6	10,728	18,375	64	0,139	0,4901	1,0821	3	5,9613	10,391	17,727
	100	0,0943	0,8673	1,368	3,4077	7,1063	10,686	14,451	100	0,0763	0,6783	1,3661	3,3138	6,8372	10,41	12,796



Για τις αναγνώσεις (70%):

- Ο συνολικός χρόνος αυξάνεται αναλογικά με τον αριθμό των operations, αλλά αυτή η αύξηση δεν είναι γραμμική λόγω της ταυτόχρονης εκτέλεσης των πολλών αναγνωστών.
- Για μεγάλο αριθμό operations (≥ 500.000), ο συνολικός χρόνος αρχίζει να αυξάνεται σημαντικά, κυρίως εξαιτίας του αυξημένου overhead που προκύπτει από την ανάγκη για μεγαλύτερη αναμονή των αναγνωστών, λόγω των λίγων αλλά συχνών εγγραφών.

Για τις εγγραφές (30%):

- Παρόμοια συμπειριφορά παρατηρείται και εδώ: όσο αυξάνεται το πλήθος των εργασιών, αυξάνεται και ο συνολικός χρόνος εκτέλεσης, κυρίως λόγω των compactions και της ανάγκης αποκλειστικής πρόσβασης.
- Στις περιπτώσεις μεγάλου αριθμού threads (≥ 32 threads), ο συνολικός χρόνος είναι σχετικά μικρότερος για μικρό πλήθος εργασιών, κάτι που επιβεβαιώνει το πλεονέκτημα του παράλληλου concurrency για μικρά workloads. Όμως, καθώς τα operations αυξάνονται, ο συνολικός χρόνος αυξάνεται έντονα λόγω του έντονου ανταγωνισμού μεταξύ threads για αποκλειστική πρόσβαση.