Πανεπιστήμιο Ιωαννίνων
Τμήμα Μηχανικών Η/Υ & Πληροφορικής

Ακαδημαϊκό Έτος 2025 -2026
Χειμερινό εξάμηνο

ΜΥΕ029
Προσομοίωση και Μοντελοποίηση Υπολογιστικών Συστημάτων

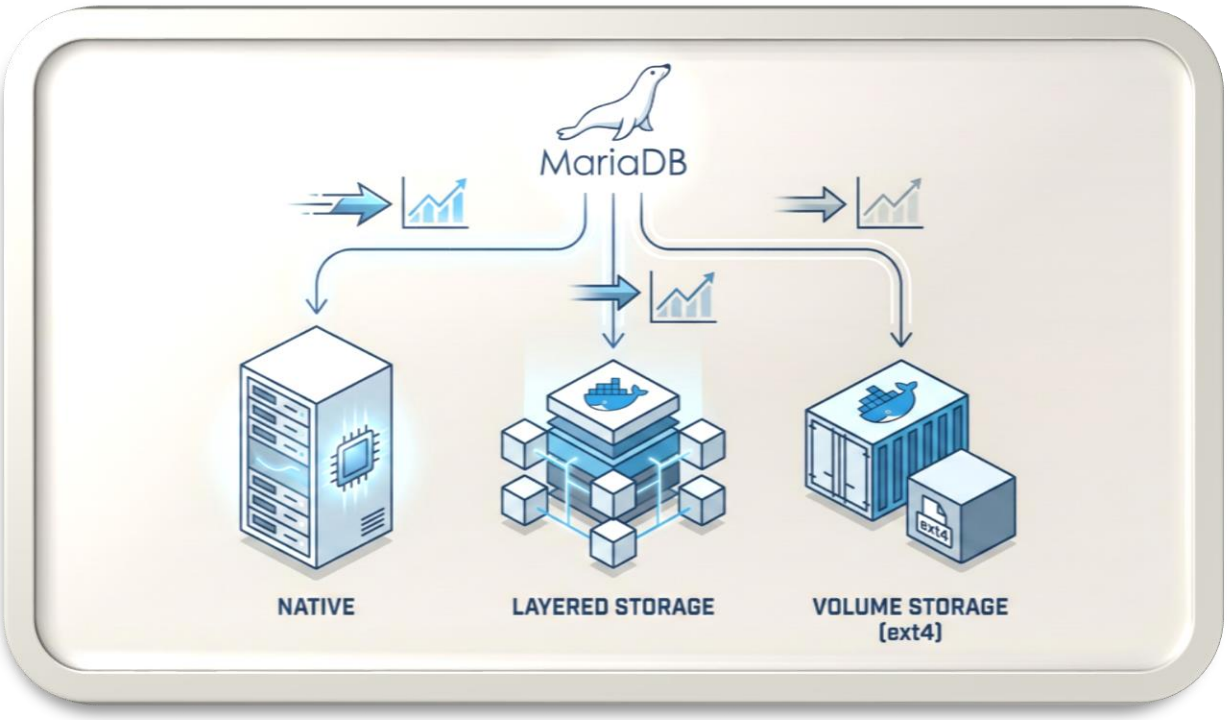
Διδάσκων: Γιώργος Καππές

1η Εργαστηριακή Άσκηση

Ανάλυση της απόδοσης του Συστήματος Διαχείρισης Βάσεων Δεδομένων MySQL σε περιβάλλον εικονικοποίησης

Ομάδα:
Παναγιώτης Παρασκευόπουλος ΑΜ:2905
Άγγελος Μπεζαΐτης ΑΜ:4432

Περιεχόμενα



Εισαγωγή

Στην παρούσα αναφορά παρουσιάζουμε τη μεθοδολογία και τα αποτελέσματα της πειραματικής αξιολόγησης της απόδοσης του Συστήματος Διαχείρισης Βάσεων Δεδομένων MariaDB. Μας ενδιέφερε κυρίως να δούμε πώς επηρεάζει την απόδοση η εκτέλεση της βάσης μέσα σε containers (εικονικοποίηση επιπέδου λειτουργικού συστήματος).

Συγκεκριμένα, μελετήθηκαν και συγκρίθηκαν τρία διαφορετικά σενάρια εκτέλεσης:

Native: Εκτέλεση της βάσης απευθείας στο σύστημα φιλοξενίας (Host OS).

Docker Layered: Εκτέλεση εντός περιέκτη Docker, με αποθήκευση δεδομένων στο σύστημα αρχείων πολλαπλών επιπέδων (Overlay2).

Docker Volume: Εκτέλεση εντός περιέκτη Docker, με αποθήκευση δεδομένων σε ειδικό, ξεχωριστό τόμο (Volume) συστήματος αρχείων ext4.

Για την παραγωγή του φόρτου εργασίας χρησιμοποιήθηκε το εργαλείο YCSB (Yahoo! Cloud Serving Benchmark), εξετάζοντας σενάρια με διαφορετικά χαρακτηριστικούς συνδυασμούς αναγνώσεων/εγγραφών (Workloads A, B, C) και διαφορετικά επίπεδα παραλληλισμού (1, 4, 8 νήματα).

Για να είμαστε σίγουροι ότι τα αποτελέσματα είναι σωστά, φτιάξαμε ένα ειδικό αυτοματοποιημένο πρόγραμμα σε γλώσσα Python. Αυτό το πρόγραμμα αναλαμβάνει να κάνει τα πειράματα ξανά και ξανά, να ξεκινάει τη βάση από το μηδέν (cold start) κάθε φορά και να εξασφαλίζει στατιστική εγκυρότητα (με 95% διάστημα εμπιστοσύνης).

Τέλος, για την ερμηνεία των διαφορών στην απόδοση, πραγματοποιήθηκε ανάλυση χαμηλού επιπέδου με χρήση εργαλείων παρατήρησης του πυρήνα Linux (perf, iostat, vmstat, mpstat), εντοπίζοντας τα σημεία συμφόρησης (bottlenecks) και το κόστος (overhead) που εισάγει η εικονικοποίηση στη διαχείριση της μνήμης και των πόρων Εισόδου/Εξόδου.

Περιγραφή Πειραματικού Περιβάλλοντος

Το πειραματικό περιβάλλον εστιάζει στην ανάλυση της απόδοσης του Συστήματος Διαχείρισης Βάσεων Δεδομένων MariaDB σε περιβάλλον εικονικοποίησης. Συγκεκριμένα, αποτελείται από έναν φυσικό κόμβο φιλοξενίας και μία εικονική μηχανή πελάτη. Η εικονική μηχανή συνδέεται στο δίκτυο μέσω NAT και εκτελεί το Λειτουργικό Σύστημα Debian 11 Linux.

Ο κόμβος φιλοξενίας διαθέτει επεξεργαστή AMD Ryzen 5 5600 6-Core 3.5GHz, 16GB DDR4 RAM με συχνότητα 3200MHz και λειτουργικό σύστημα Windows 11 Pro 24H2. Για αποθήκευση, διαθέτει 1 δίσκο SSD Samsung 990 PRO 1TB M.2 NVMe (Read Speed 7450 MB/s, Write Speed 6900 MB/s). Χρησιμοποιεί το λογισμικό VMWare Workstation Pro για την υποστήριξη της εικονικοποίησης και τη φιλοξενία της εικονικής μηχανής.

Η εικονική μηχανή διαθέτει 1 πυρήνα επεξεργαστή και 2GB μνήμης RAM. Για αποθήκευση, διαθέτει έναν εικονικό δίσκο μέγιστης χωρητικότητας 15GB, ο οποίος περιλαμβάνει μια ξεχωριστή διαμέριση (`/dev/sda3`) μεγέθους 4.1 GB. Η διαμέριση αυτή χρησιμοποιείται αποκλειστικά για την αποθήκευση των δεδομένων της βάσης MariaDB στα σενάρια Native και Volume. Στο σύστημα είναι εγκατεστημένο το λογισμικό MariaDB, το Docker για την υποστήριξη των σεναρίων Layered και Volume, καθώς και το εργαλείο YCSB (v0.17.0).

Υλοποίηση της Εργασίας

Βήμα 1 - Προετοιμασία περιβάλλοντος

Πρώτο μας βήμα, ήταν να ετοιμάσουμε την εικονική μηχανή (VM) για τα πειράματά μας, αλλάζοντας κρίσιμες ρυθμίσεις του πυρήνα (kernel) που αφορούν τον τρόπο που γράφονται τα δεδομένα στο δίσκο.
Σκοπός της άσκησης είναι να μετρήσουμε την απόδοση του δίσκου κάτω από τρία σενάρια.
Το Linux, για να είναι γρήγορο, δεν γράφει τα δεδομένα αμέσως στο δίσκο. Τα κρατάει προσωρινά στη γρήγορη μνήμη RAM (Page Cache). Τα δεδομένα που έχουν αλλάξει στη RAM αλλά δεν έχουν γραφτεί ακόμα στο δίσκο ονομάζονται "dirty".
Οπότε, αφού είχαμε συνδεθεί ως root, εκτελέσαμε τις εντολές:
echo 500 > /proc/sys/vm/dirty_expire_centisecs
echo 100 > /proc/sys/vm/dirty_writeback_centisecs
(στην εκφώνηση της άσκησης υπάρχει τυπογραφικό στο όνομα των αρχείων).
Με την πρώτη εντολή (dirty_expire_centisecs=500), λέμε στον πυρήνα: "Δεν θέλω να κρατάς 'dirty' δεδομένα στη μνήμη για πάνω από 5 δευτερόλεπτα (500 centiseconds). Μετά από 5 δευτερόλεπτα, πρέπει να τα γράψεις στο δίσκο."
Με τη δεύτερη εντολή (dirty_writeback_centisecs=100), λέμε: "Θέλω τα νήματα του πυρήνα που ελέγχουν πότε θα γράψουν τα δεδομένα στο δίσκο, να 'ξυπνάνε' κάθε 1 δευτερόλεπτο (100 centiseconds) για να κάνουν αυτή τη δουλειά."
Αν δεν το κάναμε αυτό, το YCSB θα έτρεχε τα πειράματα, θα έγραφε τα πάντα στη γρήγορη RAM και θα μας έδινε ψεύτικα, εξαιρετικά γρήγορους χρόνους. Εμείς αναγκάσαμε το σύστημα να γράφει τα δεδομένα στον αργό δίσκο, ώστε να μπορέσουμε να μετρήσουμε την πραγματική διαφορά απόδοσης μεταξύ native, layered και volume.

Βήμα 2 - Επιλογή Παραγόντων και Επιπέδων

Πριν αρχίσουμε να τρέχουμε πειράματα στα τυφλά, έπρεπε να φτιάξουμε ένα σχέδιο. Αυτό το σχέδιο καθορίζει τι ακριβώς θα αλλάζουμε σε κάθε πείραμα και τι θα μετράμε. Αυτές οι μεταβλητές που αλλάζουμε ονομάζονται παράγοντες (factors) και οι τιμές που παίρνουν ονομάζονται επίπεδα (levels).
Η εκφώνηση μας έδινε ήδη τους δύο παράγοντες που πρέπει να εξετάσουμε: το ποσοστό αναγνώσεων και ενημερώσεων και τον αριθμό των νημάτων.

Για την επιλογή των κατάλληλων τιμών για το φορτίο εργασίας (Read/Update), σκεφτήκαμε ότι θέλαμε να δούμε πως συμπεριφέρεται η βάση δεδομένων σε διαφορετικά σενάρια χρήσης. Ένα σύστημα που κυρίως διαβάζει δεδομένα έχει άλλη συμπεριφορά από ένα σύστημα που συνέχεια γράφει νέα δεδομένα.
Το YCSB μας δίνει έτοιμα, προκαθορισμένα φορτία εργασίας, οπότε χρησιμοποιήσαμε τα τρία βασικά:
Workload A: 50% Read / 50% Update (πολλές ενημερώσεις αλλά ισορροπημένο).
Workload B: 95% Read / 5% Update (Κυρίως αναγνώσεις).
Workload C: 100% Read (Μόνο αναγνώσεις).

Επίσης, θέλαμε να δούμε πως ανταποκρίνεται το σύστημα καθώς αυξάνεται ο ταυτόχρονος φόρτος.
Η VM μας έχει 1 πυρήνα CPU, άρα η αύξηση των νημάτων έχει νόημα. Όταν ένα νήμα περιμένει να ολοκληρωθεί μια λειτουργία δίσκου, ο CPU μπορεί να εκτελέσει ένα άλλο νήμα. Έτσι, μετράμε την απόδοση του συστήματος μας σε ταυτόχρονα I/O.
Αποφασίσαμε να δοκιμάσουμε 3 επίπεδα για να δούμε αυτή τη συμπεριφορά:
1 νήμα (ως βάση, χωρίς καθόλου παραλληλισμό)
4 νήματα (μέτριος φόρτος)
8 νήματα (υψηλός φόρτος)

Άρα, έχουμε 3 (Workloads) x 3 (Thread levels) = 9 συνδυασμούς πειραμάτων. Αυτούς τους συνδυασμούς τους τρέχουμε και για τις 3 περιπτώσεις (native, layered, volume).

Βήμα 3 - Καθορισμός Μεγέθους Φορτίου

Αυτό ήταν ίσως το πιο κρίσιμο σημείο της ρύθμισης.
Ο στόχος μας ήταν να μετρήσουμε την απόδοση του δίσκου. Αν το φορτίο εργασίας μας (δηλαδή ολόκληρη η βάση δεδομένων) ήταν μικρό, τότε το MariaDB θα το χωρούσε ολόκληρο στη γρήγορη μνήμη RAM (στο Buffer Pool) και δε θα χρειαζόταν να διαβάσει/γράψει καθόλου στον δίσκο κατά τη διάρκεια του πειράματος. Αυτό θα μας έδινε εξαιρετικά γρήγορα αποτελέσματα και δεν είναι το ζητούμενο της άσκησης.
Άρα η στρατηγική μας ήταν να δημιουργήσουμε ένα φορτίο δεδομένων που είναι μεγαλύτερο από τη διαθέσιμη μνήμη cache.

Τρέχοντας την εντολή **cat /etc/mysql/my.cnf | grep innodb_buffer_pool_size** στο τερματικό μπορέσαμε να δούμε πόση μνήμη έχει ρυθμιστεί για το Buffer Pool της MariaDB.
Η απάντηση **innodb_buffer_pool_size = 134217728** μας λέει ότι η μνήμη τελικά είναι ακριβώς 128 MB.

Οπότε στοχεύσαμε σε ένα συνολικό μέγεθος βάσης δεδομένων περίπου **512 MB**, το οποίο είναι 4 φορές μεγαλύτερο από το Buffer Pool, επομένως σίγουρα προκαλούμε κίνηση στο δίσκο και επίσης είναι πολύ μικρότερο από το όριο της διαμέρισης /dev/sda3 (4.1 GB) άρα δεν θα έχουμε πρόβλημα χώρου.
Για να το πετύχουμε αυτό, ρυθμίσαμε το YCSB να δημιουργεί **500.000 πλειάδες** (records).
Ανοίξαμε τα αρχεία workloada, workloadb και workloadc που χρησιμοποιήσαμε και μέσα σε καθένα από αυτά, αλλάξαμε τις τιμές των recordcount και operationcount σε:
recordcount = 500000
operationcount = 500000

Το recordcount λέει στο YCSB να δημιουργήσει 500 χιλιάδες πλειάδες κατά τη φάση load. Αυτό φτιάχνει τη βάση μας του 512 MB.
Το operationcount λέει στο YCSB να εκτελέσει 500 χιλιάδες λειτουργίες (Reads ή Updates) κατά τη φάση run. Αυτό εξασφαλίζει ότι το τεστ μας είναι αρκετά μεγάλο για να πάρουμε σταθερές μετρήσεις.

Βήμα 4 - Αυτοματοποίηση και Εκτέλεση Πειραμάτων.

Αφού ήμασταν έτοιμοι με τον σχεδιασμό, έχουμε ρυθμίσει το cache, έχουμε ορίσει τα επίπεδα (Workloads A,B,C και Threads 1,4,8) και έχουμε ρυθμίσει το μέγεθος (500K records), προχωρήσαμε στο επόμενο βήμα που ήταν το πιο σύνθετο αλλά και το πιο σημαντικό.

Έχουμε 9 συνδυασμούς πειραμάτων (3 workloads x 3 επίπεδα threads) και 3 σενάρια αποθήκευσης (native, layered, volume). Αυτό μας κάνει 27 μοναδικά πειράματα.

Όπως έχουμε δει και στις διαφάνειες του μαθήματος, μία μόνο μέτρηση δεν είναι ποτέ αξιόπιστη. Η εκφώνηση ζητά να επαναλάβουμε κάθε πείραμα (τουλάχιστον 3 φορές) και να υπολογίσουμε τη μέση τιμή και το **95% διάστημα εμπιστοσύνης (95% CI)**. Αν το διάστημα εμπιστοσύνης είναι πολύ μεγάλο (π.χ. > 5% της μέσης τιμής), πρέπει να συνεχίσουμε τις επαναλήψεις.

Το να κάναμε 27 πειράματα * 3+ επαναλήψεις (δηλαδή 81+ εκτελέσεις) με το χέρι ήταν αδύνατο και επιρρεπές σε λάθη. Γι' αυτό, φτιάξαμε ένα script σε Python που τα κάνει όλα αυτόματα.
Για τη δημιουργία αυτού του script, βασιστήκαμε για αρχή στον ψευδοκώδικα του παραρτήματος 1 της εκφώνησης και στην συνέχεια το επεκτείναμε, προσθέτοντας ότι

χρειαζόμασταν για να είναι σωστή η υλοποίηση μας. Παρακάτω αναλύουμε τα κομμάτια του κώδικα, αν και τα σχόλια μέσα σε αυτόν, νομίζω είναι αρκετά αναλυτικά.

Σημείωση: Η παρακάτω ανάλυση εστιάζει στον τεχνικό σχεδιασμό του κώδικα - τι κάνει κάθε κομμάτι και γιατί το υλοποιήσαμε έτσι. Αν δεν σας ενδιαφέρει, προχωρήστε στην Ανάλυση των αποτελεσμάτων.

Πρώτα από όλα να πούμε ότι εγκαταστήσαμε τις βιβλιοθήκες numpy, για να κάνει τους στατιστικούς υπολογισμούς (όπως η μέση τιμή και η τυπική απόκλιση) και scipy την οποία χρειαζόμασταν για να υπολογίσουμε την τιμή t-value για το διάστημα εμπιστοσύνης.

Ξεκινήσαμε την υλοποίηση του script ορίζοντας τους παράγοντες του πειράματος

```
# --- 1. ΟΡΙΣΜΟΣ ΠΑΡΑΓΟΝΤΩΝ ΚΑΙ ΕΠΙΠΕΔΩΝ ---
DB_SCENARIOS = ["native", "layered", "volume"]
WORKLOADS = ["workloada", "workloadb", "workloadc"]
THREADS = [1, 4, 8]

# --- 2. ΡΥΘΜΙΣΕΙΣ ΓΙΑ ΤΟ CI ---
MIN_RUNS = 3 # Ελάχιστες επαναλήψεις
ALPHA = 0.05 # Για 95% διάστημα εμπιστοσύνης
CI_THRESHOLD = 0.05 # Σταματάμε αν το (CI / mean) < 5%
MAX_RUN_ATTEMPTS = 8
```

run_command

Φτιάξαμε την συνάρτηση run_command, η οποία είναι μια διεπαφή για όλες τις αλληλεπιδράσεις με το shell.

```
def run_command(command, cwd=None, suppress_errors=False, capture_output=False):
    """Εκτέλεση εντολών"""
    effective_cwd = cwd or os.getcwd()

    if not suppress_errors:
        print(f"--- EXEC: {command}")

    try:
        if command.strip().endswith("&"):
            """Χρειαζόμαστε shell=True για εντολές όπως 'echo 3 > ...' και pkill"""
            subprocess.Popen(command, shell=True, cwd=effective_cwd,
                             stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
            time.sleep(0.5)
            return None
        elif capture_output:
            return subprocess.run(command, shell=True, cwd=effective_cwd,
                                  capture_output=True, text=True, timeout=300)
        else:
            subprocess.run(command, shell=True, check=True, cwd=effective_cwd,
                           stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
    except subprocess.CalledProcessError as e:
        if not suppress_errors:
            print(f"!!! ERROR: {command}")
    except Exception as e:
        if not suppress_errors:
            print(f"!!! GENERAL ERROR: {command}")

    return None
```

Με την εντολή **effective_cwd = cwd or os.getcwd()** καθορίζουμε το working directory. Αν cwd=None χρησιμοποιούμε τον τρέχοντα φάκελο ως default. Στη συνέχεια του κώδικα ρυθμίζουμε cwd=YCSB_HOME για YCSB commands που πρέπει να τρέχουν από συγκεκριμένο directory.
Ελέγχουμε το suppress_errors. Για κρίσιμες εντολές (YCSB, DB Operations) είναι False και για background processes όπως monitoring που δεν θέλουμε θόρυβο είναι True.
Χρησιμοποιούμε Popen για τα εργαλεία mpstat, iostat, vmstat, το οποίο επιτρέπει στο script να συνεχίσει την εκτέλεση (να τρέξει το benchmark) ενώ οι παρατηρητές καταγράφουν δεδομένα στο παρασκήνιο.
Χρησιμοποιούμε run για κρίσιμες εντολές (π.χ. ycsb run, docker run) όπου το script πρέπει να περιμένει την ολοκλήρωσή τους πριν προχωρήσει.
Με το capture_output ανακτούμε τα αποτελέσματα από εργαλεία όπως YCSB. Του δίνουμε και ένα timeout 5 λεπτών max για προστασία από ατέρμονες διαδικασίες.

start/stop_monitoring

Οι συναρτήσεις start_monitoring και stop_monitoring ξεκινούν και σταματούν τα εργαλεία παρακολούθησης (mpstat, vmstat, iostat).

```python
def start_monitoring(run_id, scenario, workload, threads):
    """Ξεκινάει το monitoring των πόρων συστήματος"""
    print("--- STARTING SYSTEM MONITORING ---")

    run_command(f"mkdir -p {MONITORING_DIR}", suppress_errors=True)

    base_filename = f"{MONITORING_DIR}/monitor_{scenario}_{workload}_{threads}th_run{run_id}"
    run_command(f"rm -f {base_filename}.*", suppress_errors=True)

    # Εκκίνηση monitoring tools
    tools = {
        'mpstat': f"mpstat 1 > {base_filename}.mpstat",
        'vmstat': f"vmstat 1 > {base_filename}.vmstat",
        'iostat': f"iostat -dx 1 > {base_filename}.iostat"
    }

    monitoring_pids = {}
    for tool, cmd in tools.items():
        proc = subprocess.Popen(f"{cmd} &", shell=True,
                                stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
        monitoring_pids[tool] = proc.pid

    time.sleep(2)
    return monitoring_pids, base_filename

def stop_monitoring(monitoring_pids):
    """Τερματίζει όλα τα monitoring processes"""
    print("--- STOPPING SYSTEM MONITORING ---")

    for tool, pid in monitoring_pids.items():
        try:
            os.kill(pid, signal.SIGTERM)
            time.sleep(1)
            try:
                os.kill(pid, 0)
                os.kill(pid, signal.SIGKILL)
                print(f"--- Killed {tool} process (PID: {pid})")
            except OSError:
                pass  # Process already terminated
        except (OSError, ProcessLookupError):
            pass  # Process already gone

    # Επιπλέον καθαρισμός
    for tool in ['mpstat', 'vmstat', 'iostat']:
        run_command(f"pkill -f '{tool} 1'", suppress_errors=True)

    time.sleep(2)
```

Η λειτουργία του start_monitoring είναι η εξής:
Δημιουργούμε τον φάκελο monitoring, αν δεν υπάρχει.
Ορίζουμε το όνομα των αρχείων που θα αποθηκευτούν, έτσι ώστε μόνο από το filename να μπορούμε να γνωρίζουμε ποιο πείραμα τρέχει, πόσες επαναλήψεις έχουν γίνει και ποιες μετρικές θα βρούμε μέσα.

Έπειτα δημιουργούμε το matrix των εργαλείων παρακολούθησης:
mpstat 1: Μετράει στατιστικά για τη χρήση του επεξεργαστή και τον εντοπισμό πιθανών σημείων συμφόρησης.

```
Linux 5.10.0-11-amd64 (mye029vm)        11/19/2025      _x86_64_        (1 CPU)

02:22:21 AM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
02:22:22 AM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
02:22:23 AM  all    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
02:22:24 AM  all   67.00    0.00   32.00    0.00    0.00    1.00    0.00    0.00    0.00    0.00
02:22:25 AM  all   56.44    0.00   42.57    0.00    0.00    0.99    0.00    0.00    0.00    0.00
```

vmstat 1: Μετράει στατιστικά για την χρήση της μνήμης, των διεργασιών, την εναλλαγή από δίσκο σε RAM και το αντίστροφο κ.α.

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0 243240 198144  69600 1273720    8   95   403 55736 2749 4474 12 51 33  4  0
 1  0 243240 198160  69600 1273764    0    0     0     0  181  339  0  0 100  0  0
 2  0 243240 197868  69600 1273772    0    0     0     0  174  291  0  0 100  0  0
10  0 243240 124156  69600 1275544    0    0  1600  2876 1107 34698 66 34  0  0  0
11  0 243240  76832  69600 1276272    0    0     0  7628 2053 61838 57 43  0  0  0
```

iostat 1: Αναλύει την απόδοση των δίσκων, μετρώντας στατιστικά των read/writes, το throughput, τη χρήση της CPU κ.α.

```
Linux 5.10.0-11-amd64 (mye029vm)        11/19/2025      _x86_64_        (1 CPU)

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz  %util
sda             15.77    387.66     2.13  11.92    0.09    24.58 2993.80  55182.68    88.46   2.87    0.24    18.43    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.72  57.48
sdb              0.05      3.01     5.09  99.06    1.03    62.84    0.00      0.00     0.00   0.00    1.00     0.50    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.00   0.00

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz  %util
sda              0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.00   0.00
sdb              0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.00   0.00

Device            r/s     rkB/s   rrqm/s  %rrqm r_await rareq-sz     w/s     wkB/s   wrqm/s  %wrqm w_await wareq-sz     d/s     dkB/s   drqm/s  %drqm d_await dareq-sz     f/s f_await  aqu-sz  %util
sda              0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.00   0.00
sdb              0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00      0.00     0.00   0.00    0.00     0.00    0.00    0.00    0.00   0.00
```

Στο επόμενο κομμάτι κώδικα χρησιμοποιούμε πάλι Popen και ουσιαστικά του λέμε “Ξεκίνα όλα τα εργαλεία παρακολούθησης παράλληλα”. Του δίνουμε και 2 δευτερόλεπτα χρόνο για να ξεκινήσουν τα εργαλεία και να αρχίσουν να γράφουν στα αρχεία.

Για το stop_monitoring υλοποιήσαμε ένα κομμάτι κώδικα δύο σταδίων (γιατί αρχικά υπήρχε πρόβλημα με τον τερματισμό των διεργασιών).
Στο πρώτο στάδιο, χρησιμοποιήσαμε το SIGTERM, το οποίο επιτρέπει στις διεργασίες να κάνουν cleanup, να κλείσουν τα αρχεία, να γίνει flush στους buffers και να τερματίσει όμορφα.
Στο δεύτερο στάδιο, ελέγχουμε αν η διεργασία υπάρχει ακόμα και στέλνουμε SIGKILL, για άμεσο τερματισμό.
Τέλος κάνουμε έναν επιπλέον καθαρισμό για να μην υπάρχουν τα λεγόμενα “zombie processes”.

aggresive_cleanup

Η υλοποίηση της συνάρτησης aggresive_cleanup είναι αυτή που μας έλυσε το μεγαλύτερο πρόβλημα που είχαμε στην εκτέλεση των πειραμάτων μας.

Έχοντας τελειοποιήσει το script να κάνει ακριβώς τις λειτουργίες που θέλουμε, εκτελώντας αυτόματα τα πειράματα, το τρέξαμε. Αφού έχουν εκτελεστεί τα πειράματα επιτυχώς για το σενάριο native, η εκτέλεση συνεχίζεται για το layered. Σε κάποιο σημείο σε αυτό το μέρος (δεν ήταν συγκεκριμένο σε κάθε εκτέλεση του script, για αυτό μας προβλημάτισε αρκετά), εμφάνιζε error μετά από κάποια εκτέλεση ενός πειράματος (π.χ.layered, workloadb, 1 threads) χωρίς να καταγράφει τίποτα και από αυτό το σημείο και μετά η εκτέλεση συνεχιζόταν σε έναν ατέρμονα βρόχο χωρίς να εκτελείται κανένα από τα επόμενα πειράματα.

Κάναμε αρκετές διορθώσεις του κώδικα και καταναλώσαμε πολλές ώρες ψάχνοντας και περιμένοντας το script να εκτελεστεί μετά από κάθε διόρθωση, για να μπορέσουμε να καταπολεμήσουμε αυτό το πρόβλημα και να συνεχίσει η εκτέλεση μας κανονικά.
Εν τέλει, καταλήξαμε στο συμπέρασμα ότι το πρόβλημα βρισκόταν στο ότι ο /dev/sda2 δεν καθάριζε καλά (παρόλο που είχαμε δώσει εντολές καθαρισμού μετά από την εκτέλεση κάποιων πειραμάτων) και επειδή μετά από κάποια στιγμή, ο χώρος αποθήκευσής του γέμιζε στο 100%, τα docker containers αποτύγχαναν να ξεκινήσουν, όπως και ο YCSB.

Οπότε υλοποιήσαμε την συνάρτηση aggressive_cleanup με την οποία το αντιμετωπίσαμε, κάνοντας έναν γενικό, δυνατό καθαρισμό.

```python
def aggressive_cleanup():
    """Επιθετικός καθαρισμός - ΔΙΑΤΗΡΗΣΗ ΑΠΟΤΕΛΕΣΜΑΤΩΝ"""
    print("--- SMART CLEANUP ---")

    # Διατήρηση αποτελεσμάτων, διαγραφή προσωρινών
    run_command("find /home/user/ycsb_results/ -type f ! -name '*.csv' ! -name 'monitoring' ! -path '*/monitoring/*' -delete",
                suppress_errors=True)

    # Καθαρισμός Docker και system
    run_command("docker system prune -a -f --volumes", suppress_errors=True)
    run_command("sudo apt clean", suppress_errors=True)
    run_command("sudo rm -rf /tmp/* /var/tmp/*", suppress_errors=True)

    # Τερματισμός monitoring processes
    for tool in ['mpstat', 'vmstat', 'iostat']:
        run_command(f"pkill -f '{tool} 1'", suppress_errors=True)

    # Backup αποτελεσμάτων
    run_command(f"cp {RESULTS_DIR}/final_results.csv /home/user/final_results_backup.csv 2>/dev/null",
                suppress_errors=True)

    # Έλεγχος χώρου
    result = run_command("df -h / | tail -1", capture_output=True)
    if result and result.stdout:
        print(f"--- DISK SPACE: {result.stdout.strip()}")
```

Με την πρώτη run_command του λέμε:

find /home/user/ycsb_rsults:	Βρες και άνοιξε το directory,
-type f:	μόνο τα αρχεία,
! -name '*.csv':	κράτα τα CSV αρχεία,
! -name 'monitoring':	κράτα τον φάκελο monitoring

! -path '*/monitoring/*':	και οτιδήποτε βρίσκεται μέσα σε αυτόν.
-delete:	Διέγραψε τα υπόλοιπα.

Στην δεύτερη run_command εκτελούμε αναγκαστικό καθαρισμό στα docker για να εξασφαλίσουμε ότι θα είναι σε καθαρή κατάσταση για το επόμενο πείραμα. Χρησιμοποιούμε --volumes γιατί όλα τα data μας είναι προσωρινά, τα backups γίνονται σε CSV αρχεία, όχι σε docker volumes.

Με τις επόμενες δύο run_command καθαρίζουμε διάφορα downloaded packages και ότι αρχεία βρίσκονται στο tmp.

Τέλος, κάνουμε ένα ακόμα cleanup για να “σκοτώσουμε” οποιεσδήποτε διεργασίες έμειναν, δημιουργούμε backup των αρχείων που περιέχουν τα αποτελέσματα μας και επίσης βάζουμε να τυπώνεται μετά από κάθε εκτέλεση ενός πειράματος ο χώρος του δίσκου για να μπορούμε να ελέγχουμε ότι έγινε ο καθαρισμός και είναι έτοιμος για το επόμενο πείραμα.

parse_ycsb_output και calculate_ci

Οι επόμενες δύο συναρτήσεις parse_ycsb_output και calculate_ci νομίζω δεν χρειάζονται ιδιαίτερη ανάλυση.

```
131  def parse_ycsb_output(output_text):
132      """Εξαγωγή throughput από YCSB output"""
133      for line in output_text.split('\n'):
134          if "[OVERALL], Throughput(ops/sec)," in line:
135              return float(line.split(",")[2].strip())
136      return 0.0
137
138  def calculate_ci(data):
139      """
140      Υπολογίζει το 95% CI για μια λίστα δεδομένων.
141      Επιστρέφει (mean, half_width, relative_error)
142      """
143      if len(data) < 2:
144          return 0, 0, 1.0
145
146      n = len(data)
147      mean = np.mean(data)
148      std_dev = np.std(data, ddof=1)
149
150      if std_dev == 0.0 or mean == 0.0:
151          return mean, 0.0, 1.0 # Επιστρέφουμε 1.0 (100% σφάλμα) για να συνεχίσει
152
153      std_err = std_dev / np.sqrt(n)
154      # Βρίσκουμε το t-value από τον πίνακα t (Παράρτημα 2)
155      # για (n-1) βαθμούς ελευθερίας (df) και 95% CI
156      # Η scipy το κάνει αυτόματα:
157      t_value = st.t.ppf(1 - ALPHA / 2, df=n - 1)
158      ci_half_width = t_value * std_err
159      relative_error = ci_half_width / mean
160
161      return mean, ci_half_width, relative_error
```

Η μετρική που επιλέξαμε ήταν το Througput, οπότε υλοποιήσαμε την πρώτη συνάρτηση για να την εξάγουμε από τα GBs του YCSB output χωρίς να χάσουμε performance ή ακρίβεια και να την αποθηκεύσουμε στο CSV αρχείο μας.

Με την δεύτερη συνάρτηση υπολογίζουμε το 95% CI.
Διάστημα Εμπιστοσύνης είναι το εύρος στο οποίο βρίσκεται η πραγματική μέση τιμή με 95% πιθανότητα.
Half-width είναι το μισό του εύρους (π.χ. 100 ± 5 → half-width = 5)
Relative Error: Half-width / Mean (π.χ. 5/100 = 5%)
Επειδή δεν μπορούμε να υπολογίσουμε την τυπική απόκλιση με 1 μόνο δείγμα, επιστρέφουμε 100% σφάλμα για να εξαναγκάσει περισσότερες επαναλήψεις.

get_scenario_params

Στη συνέχεια φτιάξαμε την get_scenario_params η οποία επιστρέφει τις σωστές παραμέτρους για κάθε σενάριο βάσης δεδομένων.

```
184  def get_scenario_params(scenario):
185      """Επιστρέφει τις παραμέτρους για κάθε scenario"""
186      params = {
187          "native": {
188              "init": "create-native",
189              "delete": "delete-native",
190              "start": "start-native",
191              "stop": "stop-native",
192              "properties": "jdbc-binding/conf/db.properties"
193          },
194          "layered": {
195              "init": "create-layered",
196              "delete": "delete-layered",
197              "start": "start-layered",
198              "stop": "stop-layered",
199              "properties": "jdbc-binding/conf/db.container.properties"
200          },
201          "volume": {
202              "init": "create-volume",
203              "delete": "delete-volume",
204              "start": "start-volume",
205              "stop": "stop-volume",
206              "properties": "jdbc-binding/conf/db.container.properties"
207          }
208      }
209      return params[scenario]
```

Περάσαμε όλες τις παραμέτρους σε ένα λεξικό, όπου κάθε scenario έχει ίδιο interface (init/delete/start/stop), με το native να συνδέεται απευθείας στο localhost και τα layered/volume στα docker containers.

main

Η συνάρτηση main αποτελεί την κύρια λειτουργία του πειράματος και είναι υπεύθυνη για την αυτόματη εκτέλεση των πειραμάτων μας με διαφορετικά σενάρια βάσης δεδομένων, workloads και αριθμούς νημάτων. Η διαδικασία περιλαμβάνει τρία βασικά στάδια: αρχικοποίηση, εκτέλεση πειραμάτων και τελικές λειτουργίες.

```python
def main():
    """Κύρια λειτουργία του πειράματος"""
    # Αρχικός καθαρισμός και προετοιμασία
    aggressive_cleanup()
    update_workload_files()

    run_command(f"mkdir -p {RESULTS_DIR} {MONITORING_DIR}", suppress_errors=True)

    # Προετοιμασία αρχείου αποτελεσμάτων
    file_path = f"{RESULTS_DIR}/final_results.csv"
    if not os.path.exists(file_path) or os.path.getsize(file_path) == 0:
        with open(file_path, "w") as f:
            f.write("Scenario,Workload,Threads,Runs,Mean_Throughput,CI_Half_Width,Relative_Error\n")

    results_log = open(file_path, "a")
    experiment_count = 0

    # Κύριος βρόχος πειραμάτων
    for scenario in DB_SCENARIOS:
        params = get_scenario_params(scenario)

        # Επανάληψη για κάθε workload (A, B, C)
        for workload_file in WORKLOADS:

            # Επανάληψη για κάθε αριθμό νημάτων (1, 4, 8)
            for thread_count in THREADS:
                experiment_count += 1

                # Περιοδικός καθαρισμός
                if experiment_count % 2 == 0:
                    aggressive_cleanup()

                print(f"\n*** EXPERIMENT {experiment_count}: {scenario}, {workload_file}, {thread_count} threads ***")
                run_experiment(scenario, workload_file, thread_count, params, results_log)

    # Τελικές λειτουργίες
    print("\n*** SAVING FINAL RESULTS ***")
    run_command(f"cp {RESULTS_DIR}/final_results.csv /home/user/final_results_backup.csv")
    run_command(f"ls -la {RESULTS_DIR}/ {MONITORING_DIR}/")

    aggressive_cleanup()
    results_log.close()

    print(f"\n*** ALL EXPERIMENTS COMPLETED ***")
    print(f"Results: {RESULTS_DIR}/final_results.csv")
    print(f"Monitoring Data: {MONITORING_DIR}/")
    print(f"Backup: /home/user/final_results_backup.csv")
```

1. Αρχικοποίηση

Στην αρχή καθαρίζουμε το περιβάλλον μέσω της aggressive_cleanup, διασφαλίζοντας ότι δεν υπάρχουν υπολείμματα από προηγούμενα πειράματα. Στη συνέχεια, ενημερώνουμε τα workload αρχεία με τις σωστές ρυθμίσεις. Δημιουργούμε τους απαραίτητους φακέλους για τα αποτελέσματα και τα δεδομένα παρακολούθησης, ενώ προετοιμάζουμε και το αρχείο αποτελεσμάτων final_results.csv. Αν το αρχείο δεν υπάρχει ή είναι κενό, γράφεται μια επικεφαλίδα με τα ονόματα των στηλών.

2. Εκτέλεση πειραμάτων

Ο κύριος βρόχος των πειραμάτων εκτελείται για κάθε συνδυασμό (scenario-workload-threads).

Πρώτα όλοι οι συνδυασμοί workload*threads για native, μετά για layered και μετά για volume.

Για κάθε συνδυασμό, αυξάνεται ο μετρητής πειραμάτων (experiment_count). Κάθε δύο πειράματα εκτελείται περιοδικός καθαρισμός (aggressive_cleanup) για τη διατήρηση ενός καθαρού περιβάλλοντος. Η συνάρτηση run_experiment αναλαμβάνει την εκτέλεση του πειράματος, λαμβάνοντας τις παραμέτρους του σεναρίου, το workload, τον αριθμό νημάτων και το αρχείο καταγραφής αποτελεσμάτων.

3.Τελικές λειτουργίες

Μετά την ολοκλήρωση όλων των πειραμάτων, αποθηκεύουμε τα αποτελέσματά μας σε ένα backup αρχείο σε διαφορετικό location, για να αποφύγουμε κάποια διαγραφή των αρχείων στη φάση καθαρισμού.

Κάνουμε έναν τελευταίο καθαρισμό για να αφήσουμε το σύστημα καθαρό για επόμενη χρήση και κλείνουμε το αρχείο καταγραφής.

Τέλος, εμφανίζουμε ένα μήνυμα ολοκλήρωσης μαζί με τις τοποθεσίες των αποτελεσμάτων, επιβεβαιώνοντας ότι τα αρχεία μας έχουν δημιουργηθεί.

run_experiment

Η λειτουργία της run_experiment είναι να εκτελεί όλες τις επαναλήψεις για έναν συγκεκριμένο συνδυασμό παραμέτρων.

```
261  def run_experiment(scenario, workload_file, thread_count, params, results_log):
262      """Εκτελεί ένα πείραμα με όλες τις επαναλήψεις"""
263      ycsb_workload_path = f"workloads/{workload_file}"
264
265      # Database setup
266      print("--- Setting up database...")
267      run_command(f"{SCRIPTS_HOME}/mariadb.sh {params['init']}", cwd=YCSB_HOME)
268
269      wait_time = 60 if scenario == "layered" else 40
270      print(f"Waiting {wait_time}s for DB...")
271      time.sleep(wait_time)
272
273      # YCSB Load
274      print("--- Loading data...")
275      run_command(f"./bin/ycsb.sh load jdbc -P {ycsb_workload_path} -P {params['properties']} -s",
276                  cwd=YCSB_HOME)
277
278      # Εκτέλεση επαναλήψεων
279      throughput_results = [] # Λίστα για αποθήκευση ρυθμαπόδοσης
280      total_attempts = 0
281      need_more_runs = True
282
283      while need_more_runs and total_attempts < MAX_RUN_ATTEMPTS:
284          total_attempts += 1
285          throughput = run_single_iteration(scenario, workload_file, thread_count, params, total_attempts)
286
287          if throughput > 0:
288              throughput_results.append(throughput)
289              print(f"--- Throughput: {throughput:.2f} ops/sec")
290
291              if len(throughput_results) >= MIN_RUNS:
292                  mean_throughput, ci_width, rel_err = calculate_ci(throughput_results)
293                  print(f"--- STATS: Mean={mean_throughput:.2f}, CI±{ci_width:.2f}, Error={rel_err:.2%}")
294
295                  if rel_err < CI_THRESHOLD:
296                      need_more_runs = False
297                      results_log.write(f"{scenario},{workload_file},{thread_count},{len(throughput_results)},{mean_throughput:.2f},{ci_width:.2f},{rel_err:.2%}\n")
298                      results_log.flush()
299                      break
300
301      # Cleanup μετά από κάθε experiment
302      print("--- Cleaning up...")
303      run_command(f"{SCRIPTS_HOME}/mariadb.sh {params['delete']}", cwd=YCSB_HOME)
304
305      if scenario in ["layered", "volume"]:
306          run_command("docker system prune -f", suppress_errors=True)
```

Αρχικά δημιουργούμε τη βάση δεδομένων που καθορίζεται από τις παραμέτρους του

κάθε σεναρίου. Περιμένουμε 40 δευτερόλεπτα (60 στο layered) για να διασφαλίσουμε ότι η βάση είναι έτοιμη.
Φορτώνουμε τα δεδομένα στη βάση μέσω της εντολής ycsb load και εκτελούμε τις επαναλήψεις.

Η κύρια λειτουργία της συνάρτησης είναι η εκτέλεση πολλαπλών επαναλήψεων του πειράματος. Αποθηκεύουμε τα αποτελέσματα της ρυθμαπόδοσης (throughput) σε μια λίστα. Η διαδικασία συνεχίζεται μέχρι να επιτευχθεί ο ελάχιστος αριθμός επαναλήψεων με αποδεκτό σφάλμα διαστήματος εμπιστοσύνης (CI) ή μέχρι να φτάσει τον μέγιστο αριθμό προσπαθειών.
Για κάθε επανάληψη, η συνάρτηση run_single_iteration εκτελεί το πείραμα και επιστρέφει τη ρυθμαπόδοση. Εάν η ρυθμαπόδοση είναι μεγαλύτερη από το μηδέν, προστίθεται στη λίστα αποτελεσμάτων. Όταν υπάρχουν αρκετά δεδομένα, υπολογίζονται τα στατιστικά στοιχεία (μέση τιμή, εύρος CI, σχετικό σφάλμα) μέσω της calculate_ci που υλοποιήσαμε. Αν το σφάλμα είναι μικρότερο από το όριο (CI_THRESHOLD), τα αποτελέσματα καταγράφονται στο αρχείο results_log και η διαδικασία τερματίζεται.

Μετά την ολοκλήρωση του πειράματος, διαγράφουμε την βάση δεδομένων μέσω της εντολής params['delete']. Για σενάρια όπως "layered" ή "volume", εκτελούμε επίσης καθαρισμό Docker για την απελευθέρωση πόρων.

run_single_iteration

Στη συνάρτηση run_single_iteration έχουμε υλοποιήσει τι θα κάνει κάθε επανάληψη ενός πειράματος.

```
308  def run_single_iteration(scenario, workload_file, thread_count, params, run_id):
309      """Εκτελεί μία επανάληψη του πειράματος"""
310      # Database restart
311      run_command(f"{SCRIPTS_HOME}/mariadb.sh {params['stop']}", cwd=YCSB_HOME)
312      time.sleep(3)
313      run_command(f"{SCRIPTS_HOME}/mariadb.sh {params['start']}", cwd=YCSB_HOME)
314      time.sleep(15)
315
316      # Clean caches
317      run_command("echo 3 > /proc/sys/vm/drop_caches", suppress_errors=True)
318
319      # Monitoring
320      monitoring_pids, _ = start_monitoring(run_id, scenario, workload_file, thread_count)
321
322      # YCSB Run
323      ycsb_command = f"./bin/ycsb.sh run jdbc -P workloads/{workload_file} -P {params['properties']} -threads {thread_count} -s"
324      result = run_command(ycsb_command, cwd=YCSB_HOME, capture_output=True)
325
326      stop_monitoring(monitoring_pids)
327
328      if result and result.returncode == 0:
329          return parse_ycsb_output(result.stdout)
330
331      return 0.0
```

Στην αρχή, με τις παραμέτρους stop και start, κάνουμε restart στην βάση δεδομένων. Χρησιμοποιούμε την εντολή echo 3 για να καθαρίσουμε τις κρυφές μνήμες πριν την εκτέλεση κάθε πειράματος.

Ξεκινάμε τους παρατηρητές κατευθείαν πριν το YCSB.
Εκτελούμε το YSCB με βάση το workload αρχείο, τις ιδιότητες της βάσης δεδομένων και τον αριθμό νημάτων.
Μετά την ολοκλήρωση της εκτέλεσης του YCSB, τερματίζουμε την παρακολούθηση του συστήματος μέσω της stop_monitoring. Ελέγχουμε αν η εκτέλεση του YCSB ήταν επιτυχής (επιστροφή κωδικού 0), τότε η συνάρτηση αναλύει την έξοδο του YCSB για να εξαγάγει το throughput χρησιμοποιώντας τη parse_ycsb_output, αλλιώς αν η εκτέλεση αποτύχει, επιστρέφουμε throughput 0.0.

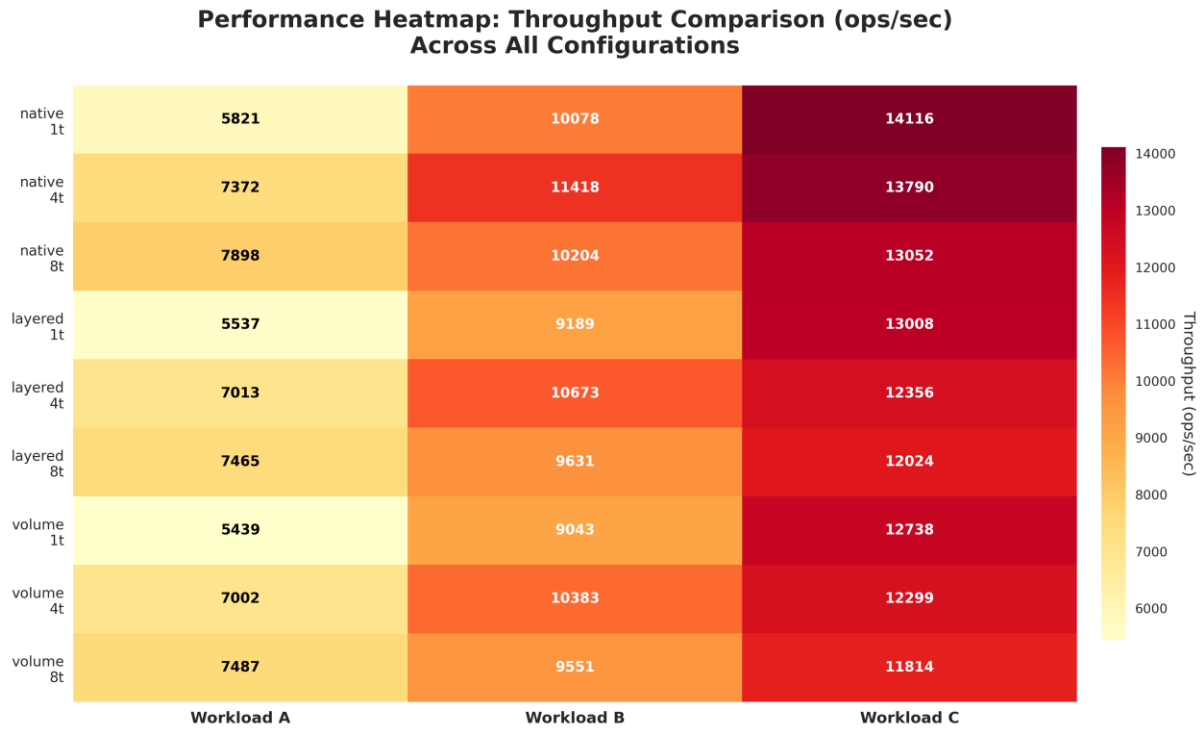
Ανάλυση αποτελεσμάτων

Στην ενότητα αυτή παρουσιάζουμε την ανάλυση της επίδοσης του συστήματος MariaDB κάτω από τα τρία εξεταζόμενα σενάρια εκτέλεσης (Native, Layered, Volume). Η αξιολόγηση βασίστηκε στη μετρική της ρυθμαπόδοσης (throughput - ops/sec) για τρία διαφορετικά φορτία εργασίας (Workloads A, B, C) και τρία διαφορετικά επίπεδα παραλληλισμού (1, 4, 8).

Για την παραγωγή των γραφικών παραστάσεων και την οπτικοποίηση των δεδομένων, χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python σε συνδυασμό με τη βιβλιοθήκη Matplotlib.

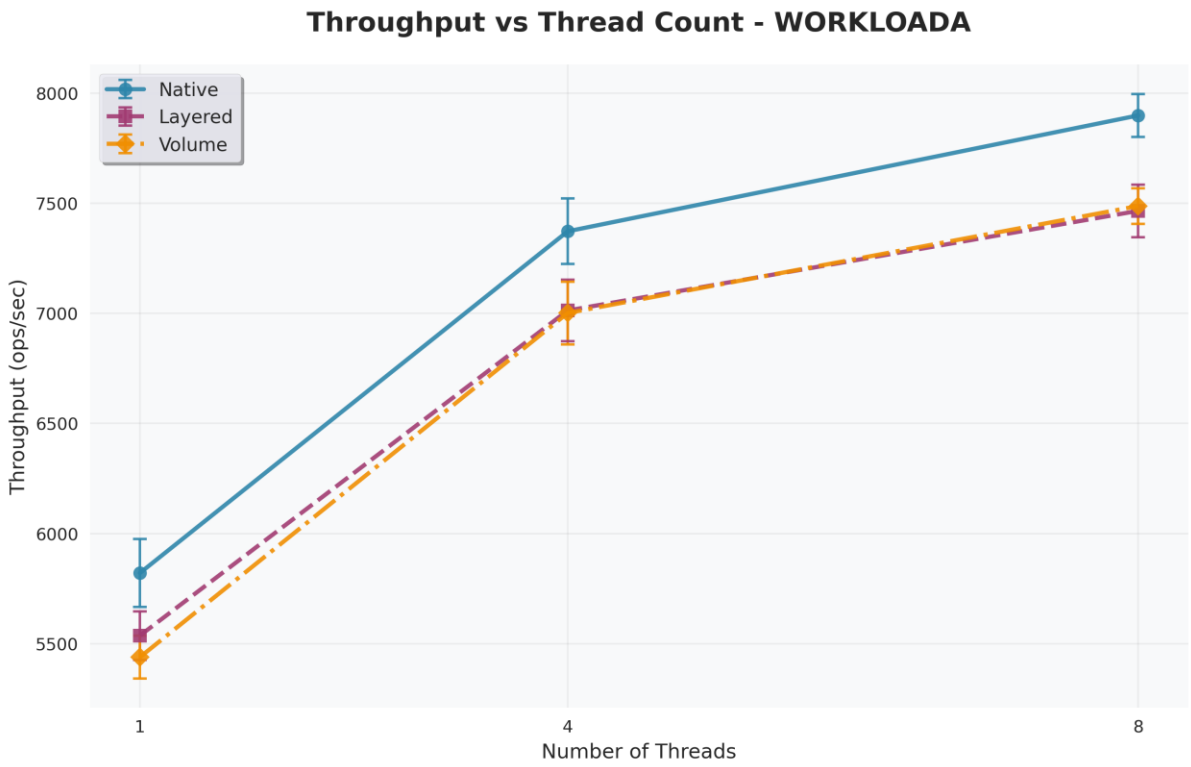
Υλοποιήσαμε συνολικά 6 scripts για κάθε ανάλυση:
Ένα για το γενικό throughput των πειραμάτων.
Δύο για perf stat και perf record/report.
Τρία για τους παρατηρητές mpstat, vmstat και iostat.

Συνολική Σύγκριση μέσω Heatmap



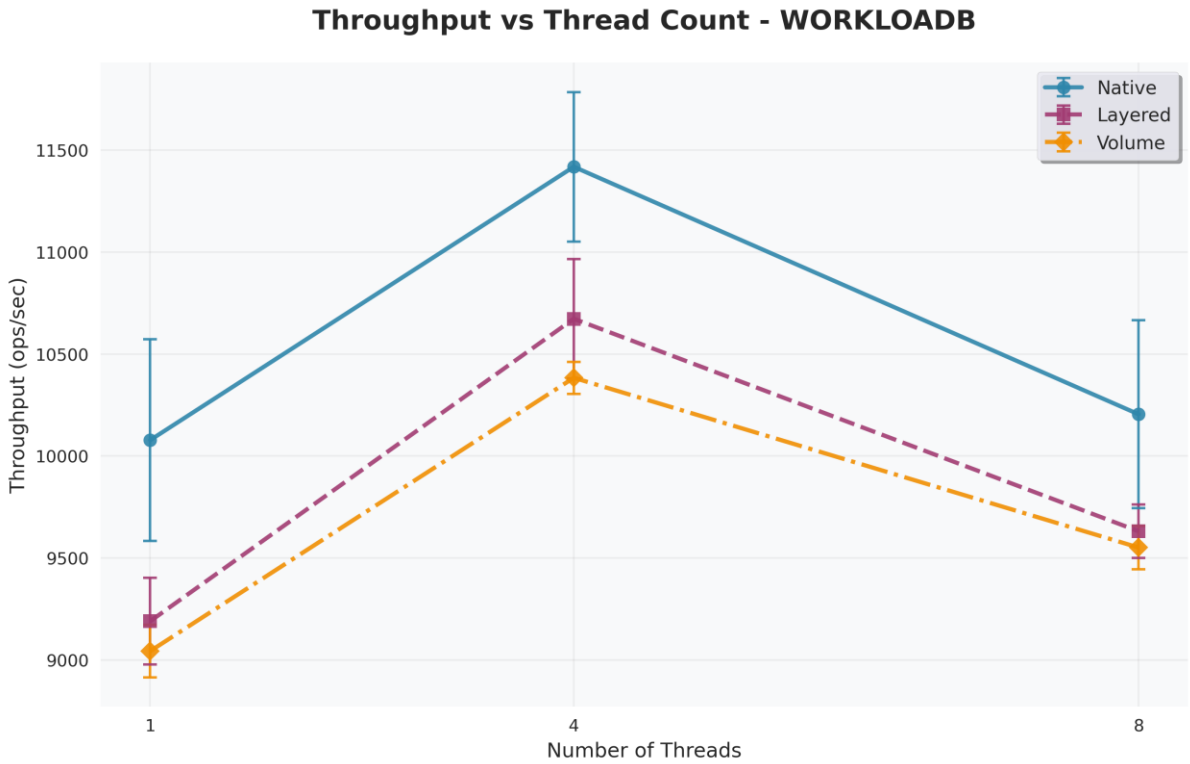
Παρατηρώντας τα αποτελέσματα, όπως φαίνονται και στο παραπάνω heatmap, είναι σαφές ότι η εκτέλεση της βάσης δεδομένων απευθείας στο σύστημα φιλοξενίας (**Native**) προσφέρει σταθερά την υψηλότερη απόδοση σε σύγκριση με τα σενάρια εικονικοποίησης, σε όλες τις συνδυαστικές περιπτώσεις workloads και threads. Τα φορτία μεγάλης έντασης αναγνώσεων (workloads B και C) επιδεικνύουν τις υψηλότερες αποδόσεις συνολικά, με το Workload C στο native deployment με 1 thread να φτάνει την απόλυτη μέγιστη απόδοση των 14,116 operations ανά δευτερόλεπτο. Η κλιμάκωση με threads φαίνεται βέλτιστη στα 4 threads για τα περισσότερα σενάρια, ενώ η διαφορά απόδοσης μεταξύ native και container deployments είναι πιο έντονη στα workloads υψηλής ανάγνωσης.

Ανάλυση Workload A: Ισορροπημένο (50% Read/ 50% Update)



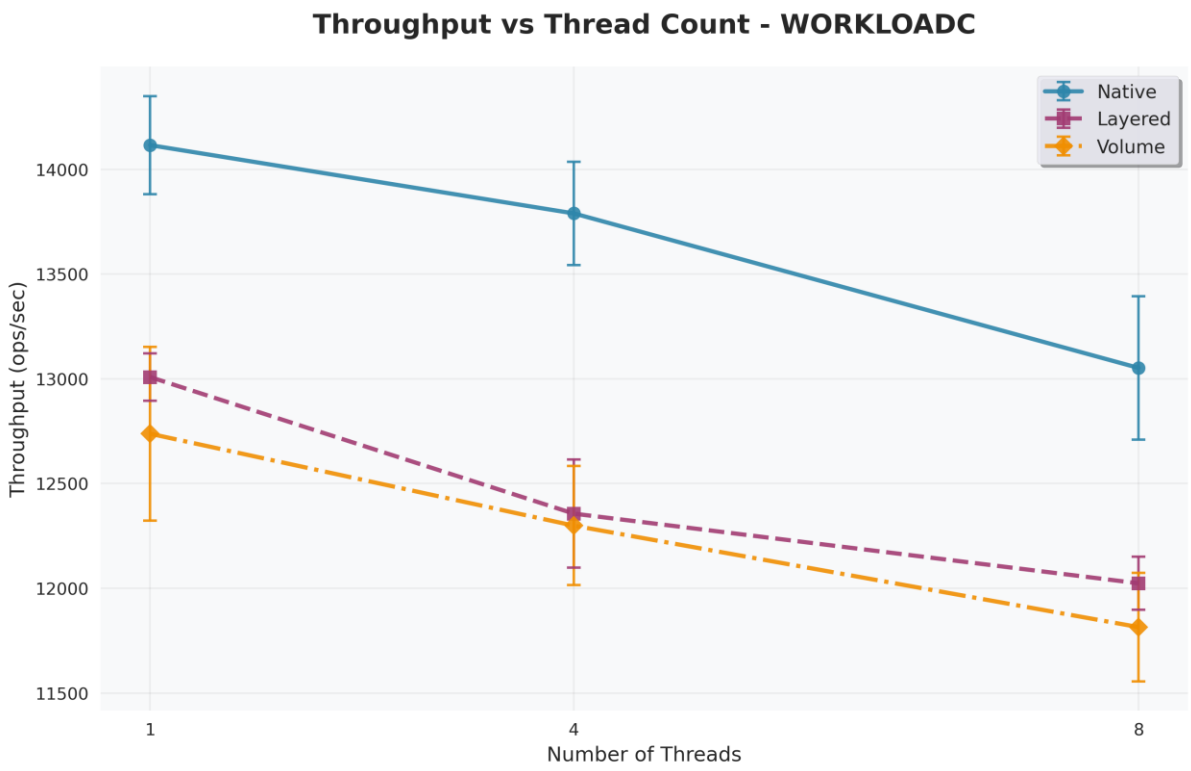
Στο Workload A, που χαρακτηρίζεται από ισορροπημένη κατανομή read/write operations (50%/50%), παρατηρούμε σταθερή κλιμάκωση απόδοσης με την αύξηση των threads και στα τρία σενάρια ανάπτυξης. Το native deployment διατηρεί σταθερή υπεροχή κατά 300-400 operations ανά δευτερόλεπτο σε κάθε επίπεδο παραλληλίας. Ενδιαφέρον παρουσιάζει η ελάχιστη διαφορά μεταξύ layered και volume, που δεν ξεπερνά το 1%, υποδεικνύοντας ότι ο τύπος του storage driver έχει αμελητέα επίδραση σε αυτό το είδος workload. Η βέλτιστη απόδοση επιτυγχάνεται στο native deployment με 8 threads (7,898 ops/sec), ενώ η επιβάρυνση του Docker παραμένει σταθερή αλλά όχι καταστροφική, με μέση μείωση απόδοσης 5-6%.

Ανάλυση Workload B: Read-Intensive (95% Read / 5% Update)



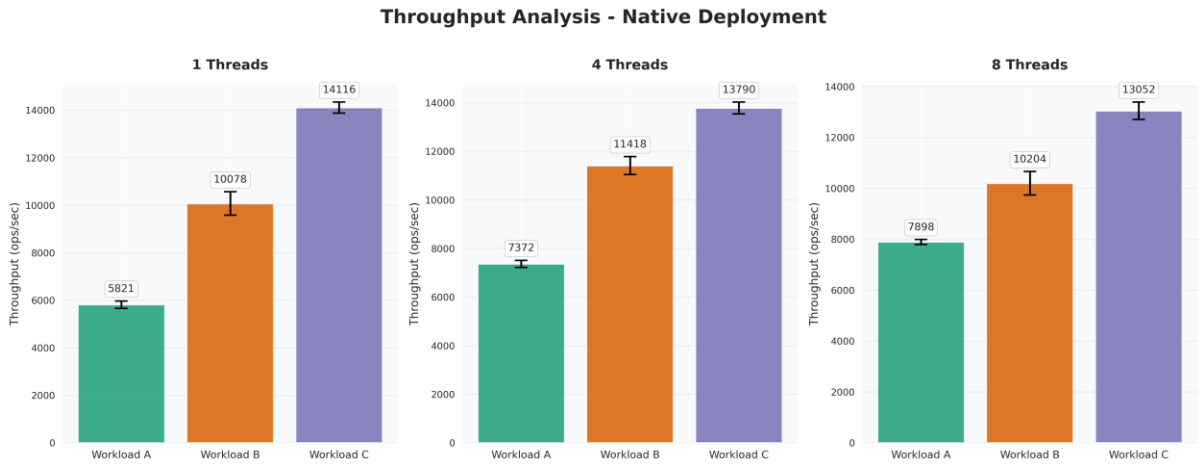
Το Workload B, που είναι read-intensive με 95% read operations, αποκαλύπτει ενδιαφέροντα μοτίβα συμπεριφοράς. Η native υλοποίηση επιτυγχάνει μέγιστη απόδοση στα 4 threads (11,418 ops/sec), ακολουθούμενη από πτώση της απόδοσης στα 8 threads, πιθανόν λόγω I/O contention. Δεδομένου ότι η εικονική μηχανή διαθέτει μόνο 1 CPU, η αύξηση των νημάτων σε εργασίες που δεν περιμένουν τον δίσκο (καθώς τα δεδομένα είναι στην cache) προκαλεί έντονο ανταγωνισμό για τον επεξεργαστή και αυξημένο κόστος εναλλαγής περιβάλλοντος (context switching), οδηγώντας σε μείωση της συνολικής απόδοσης. Σε αυτό το workload, η επιβάρυνση του Docker μεγαλώνει σημαντικά, φτάνοντας το 9% συγκριτικά με το Workload A. Αξιοσημείωτη είναι η ελαφρά υπεροχή του layered έναντι του volume σε όλα τα επίπεδα threads, καθώς και η σαφής εμφάνιση ορίου κλιμάκωσης, όπου τα 4 threads εμφανίζονται ως το βέλτιστο σημείο λειτουργίας.

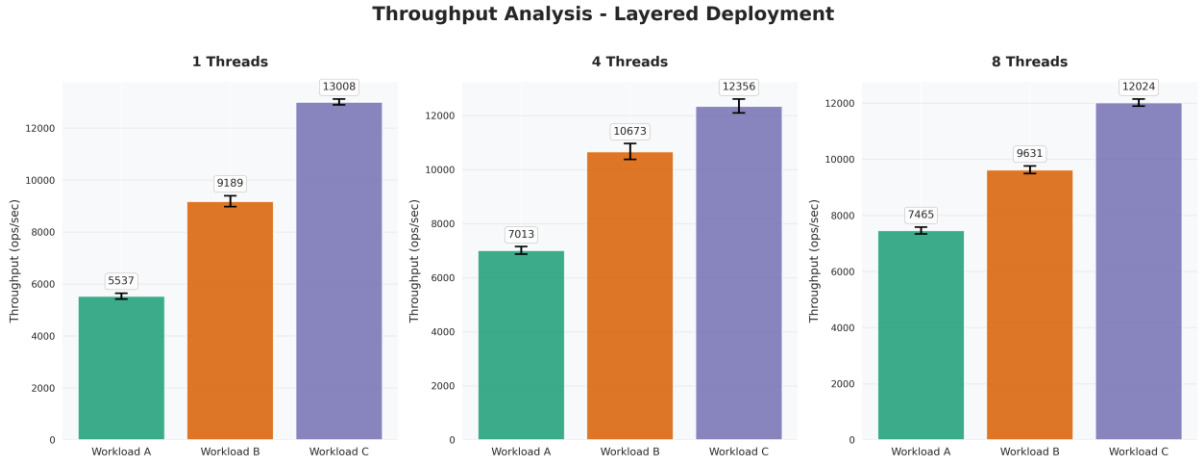
Ανάλυση Workload C: Read-only (100% Read)

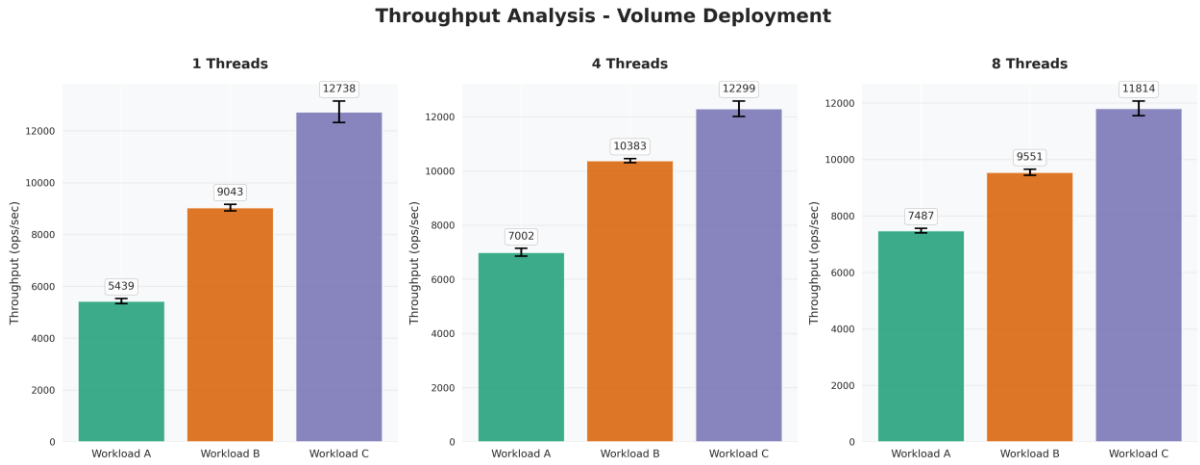


Το Workload C, ως αποκλειστικά read-only, αποκαλύπτει τα πραγματικά I/O bottlenecks του συστήματος. Το native deployment με 1 thread επιτυγχάνει την υψηλότερη απόδοση ολόκληρου του πειράματος, ακολουθούμενο από αντίστροφη κλιμάκωση όπου η απόδοση μειώνεται με την αύξηση των threads, υποδεικνύοντας I/O saturation. Σε αυτό το σενάριο, η επιβάρυνση του Docker μεγιστοποιείται, φτάνοντας το 8.7%, γεγονός που υποδηλώνει ότι τα containers επηρεάζουν περισσότερο τις καθαρές λειτουργίες ανάγνωσης. Παρατηρείται σταθερή διαφορά απόδοσης μεταξύ των υλοποιήσεων native και container σε όλα τα επίπεδα threads, επιβεβαιώνοντας ότι το workload φτάνει γρήγορα τα όρια του συστήματος και είναι ουσιαστικά δεσμευμένο από την μνήμη.

Bar Plots: Συγκριτική ανάλυση ανά Deployment Strategy







Τα bar plots παρέχουν λεπτομερή σύγκριση για κάθε στρατηγική ανάπτυξης ξεχωριστά. Στη native υλοποίηση παρατηρούμε καλύτερη απόκριση σε workloads υψηλής ανάγνωσης, ισχυρή κλιμάκωση στο Workload A, και πρόωρο κορεσμό στα read-intensive workloads. Η υλοποίηση layered επιδεικνύει παρόμοιο μοτίβο κλιμάκωσης με το native αλλά σε χαμηλότερα επίπεδα, με ιδιαίτερα καλή απόδοση σε λειτουργίες εγγραφής και ευαισθησία σε read contention. Οι τόμοι ext4 δείχνουν παρόμοια απόδοση με αυτήν του layered, με διαφορά μικρότερη του 2%, σταθερή απόδοση σε όλες τις διαφορετικές διαμορφώσεις, και καμία σημαντική πλεονεκτική θέση έναντι του layered storage για αυτό το συγκεκριμένο τύπο workload.

Συνοπτικά Συμπεράσματα

Από την ολοκληρωμένη ανάλυση προκύπτουν σαφή συμπεράσματα. Η υλοποίηση native προσφέρει 7-9% υψηλότερη απόδοση συγκριτικά με λύσεις βασισμένες σε containers, με τη διαφορά απόδοσης να μεγιστοποιείται σε read-intensive scenarios. Η επιλογή μεταξύ layered και volume storage δεν επηρεάζει σημαντικά την απόδοση, με διαφορές μικρότερες του 2%, άρα η επιλογή πρέπει να γίνει με βάση τις ανάγκες μονιμότητας των δεδομένων και όχι με κριτήριο την απόδοση. Τα 4 threads αποτελούν το βέλτιστο σημείο λειτουργίας για τα περισσότερα workloads, καθώς περισσότερα threads οδηγούν σε contention. Τα read-heavy workloads εμφανίζουν υψηλότερη ευαισθησία στην επιβάρυνση του Docker και δείχνουν πιο έντονα όρια απόδοσης. Όλα τα πειραματικά αποτελέσματα χαρακτηρίζονται από υψηλή στατιστική αξιοπιστία, με διαστήματα εμπιστοσύνης 95% και σχετικό σφάλμα μικρότερο του 5%, επιβεβαιώνοντας την εγκυρότητα των παρατηρούμενων συμπεριφορών.

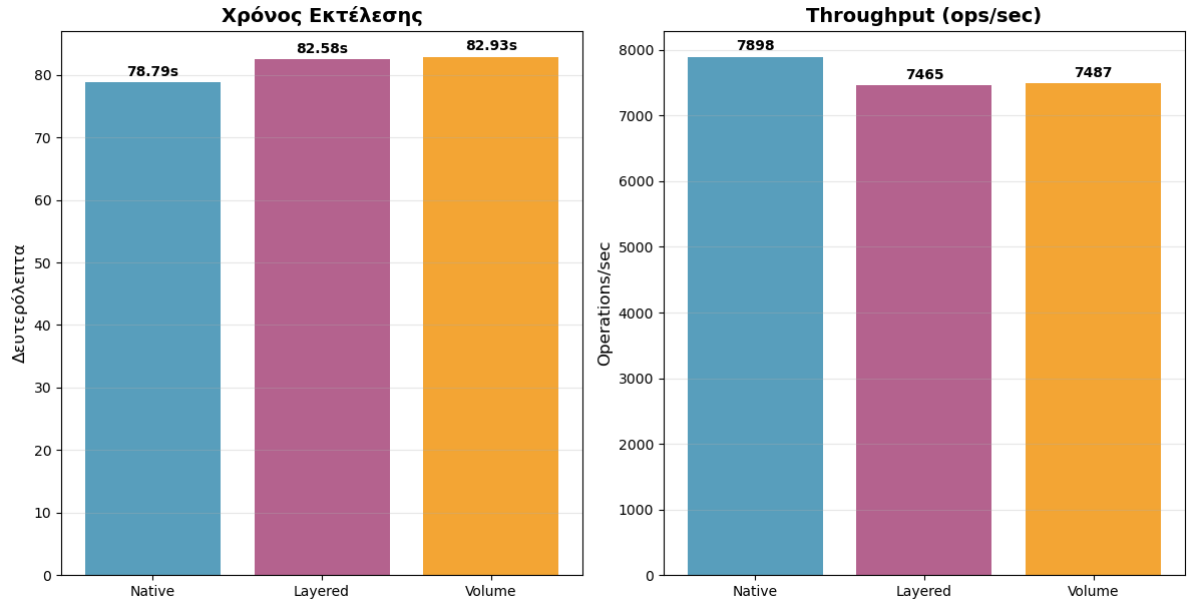
Η οπτική ανάλυση όχι μόνο επιβεβαιώνει τα νούμερα αλλά μας βοηθάει να καταλάβουμε καλύτερα πώς λειτουργούν οι εσωτερικοί μηχανισμοί, παρέχοντας μια πλήρη εικόνα της απόδοσης της MariaDB υπό διαφορετικές στρατηγικές ανάπτυξης.

Ανάλυση Επίδοσης με Χρήση του Παρατηρητή perf

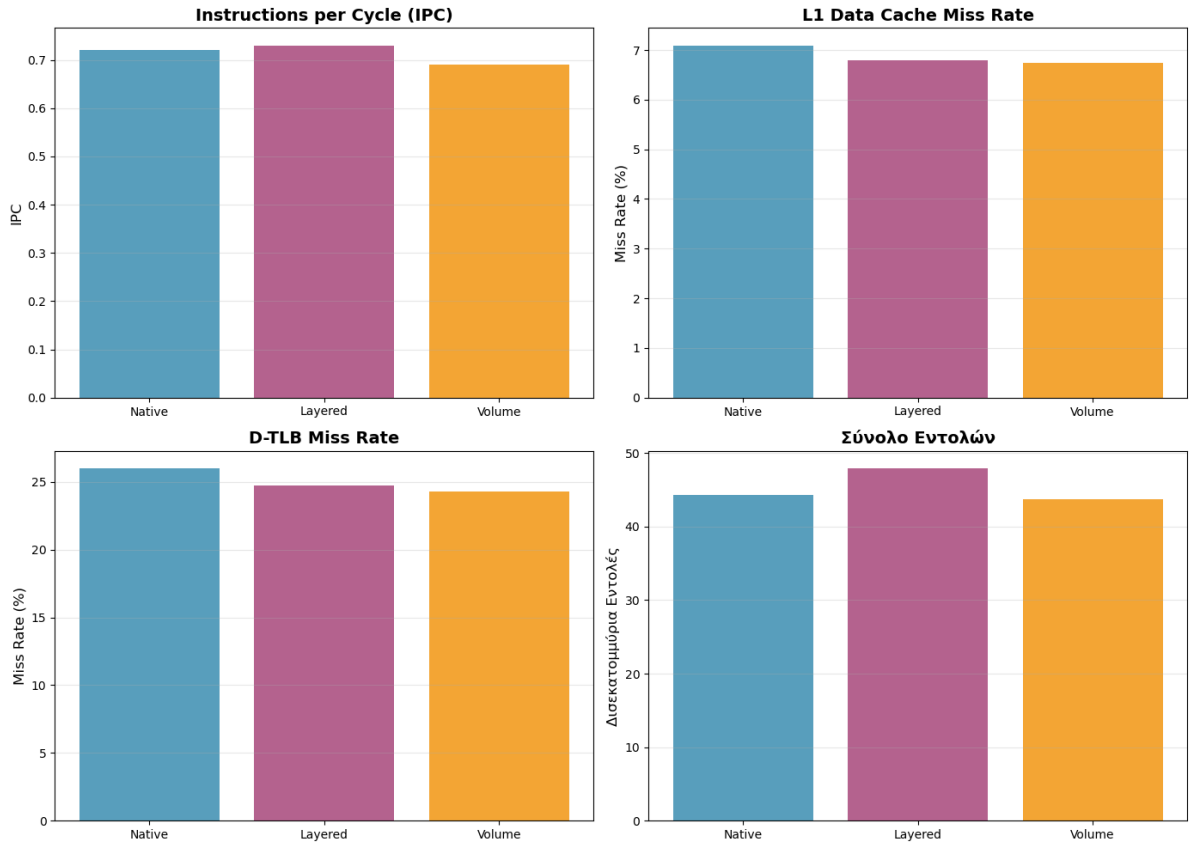
Αφού είδαμε τα “μακρο” αποτελέσματα, χρησιμοποιήσαμε το εργαλείο perf για να καταλάβουμε γιατί υπάρχουν διαφορές στην απόδοση ανάμεσα στους τρεις διαφορετικούς τρόπους ανάπτυξης, κοιτάζοντας σε “μικρο” επίπεδο.
Τα αποτελέσματα από τα perf stat και perf record μας δίνουν μια πολύ καλή εικόνα για το πώς συμπεριφέρεται το σύστημα στον επεξεργαστή, στην κρυφή μνήμη και στη μνήμη RAM.

Επιλέξαμε για σύγκριση ένα σενάριο υψηλού φόρτου όπου οι διαφορές είναι συνήθως πιο εμφανείς. Χρησιμοποιήσαμε το ίδιο φορτίο workloada (50/50 Read/Update είναι καλό για stress test), με 8 νήματα, για τους τρεις διαφορετικούς τρόπους.

Perf stat



Η υλοποίηση native επιτυγχάνει τον μικρότερο χρόνο εκτέλεσης (78,79 δευτερόλεπτα) και το υψηλότερο throughput (7.898 ops/sec), επιβεβαιώνοντας την υπεροχή της που παρατηρήθηκε και στα προηγούμενα βήματα.
Οι υλοποιήσεις βασισμένες σε containers (Layered: 82,58s, Volume: 82,93s) εμφανίζουν παρόμοιους χρόνους εκτέλεσης, με μικρή υπεροχή του layered storage έναντι του volume.



Η μετρική Instructions per Cycle (IPC) αποκαλύπτει σημαντικές διαφορές στην αποδοτικότητα του επεξεργαστή. Η υλοποίηση layered επιτυγχάνει το υψηλότερο IPC (0.73), ακολουθούμενη από την native (0.72) και την volume (0.69). Παραδόξως, το layered deployment εκτελεί περισσότερες εντολές συνολικά (47,87 δισεκατομμύρια) συγκριτικά με το native (44,26 δισεκατομμύρια), υποδηλώνοντας πρόσθετη υπολογιστική επιβάρυνσηαπό το περιβάλλον εκτέλεσης του Docker.

Η ανάλυση του συστήματος κρυφής μνήμης (cache) αποκαλύπτει ενδιαφέροντα ευρήματα. Το native deployment εμφανίζει το υψηλότερο ποσοστό αστοχίας/αποτυχίας (miss rate) της κρυφής μνήμης δεδομένων L1 (7,09%), ενώ οι υλοποιήσεις τα container deployments έχουν ελαφρώς καλύτερη απόδοση κρυφής μνήμης (Layered: 6,79%, Volume: 6,74%).

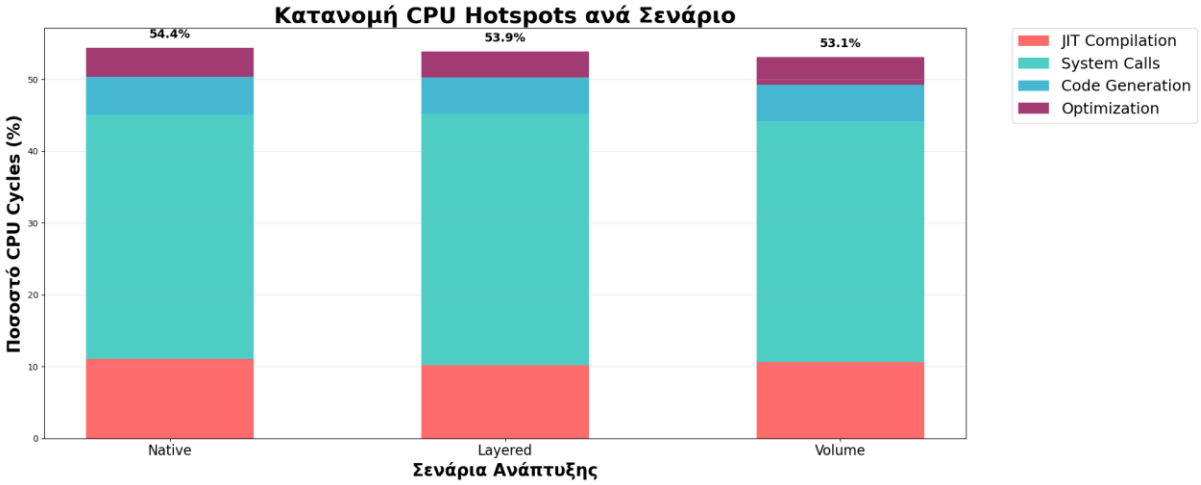
Ωστόσο, η native υλοποίηση έχει σημαντικά λιγότερα loads δεδομένων στην κρυφή μνήμη L1 (20,85 δισεκατομμύρια) συγκριτικά με το layered (23,35 δισεκατομμύρια), υποδεικνύοντας πιο αποτελεσματική χρήση της cache.

Το TLB miss rate παρουσιάζει σημαντικές διαφορές μεταξύ των deployments. Το native deployment έχει την υψηλότερη D-TLB miss rate (25.98%), ενώ τα container deployments εμφανίζουν βελτιωμένη TLB απόδοση (Layered: 24.75%, Volume: 24.31%).

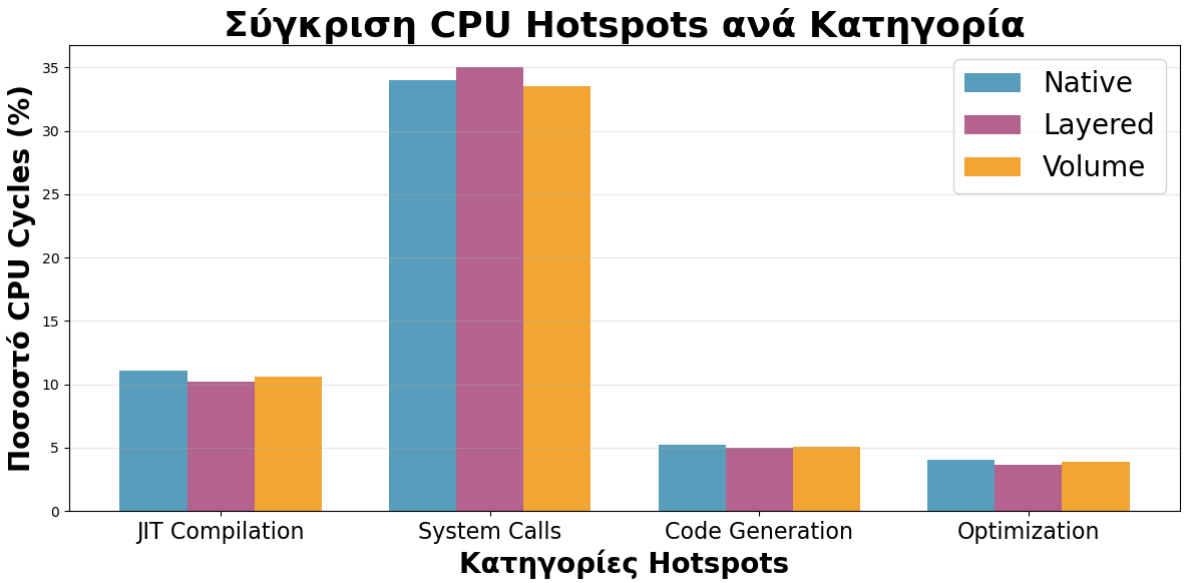
Αυτό υποδηλώνει ότι τα containers ενδέχεται να επωφελούνται από διαφορετικά μοτίβα προσπέλασης μνήμης ή από βελτιστοποιήσεις στη διαχείριση μνήμης του Docker.

Ο αριθμός των context switches είναι σχεδόν ίδιος και στα τρία σενάρια (περίπου 509,000), υποδεικνύοντας ότι η επιβάρυνση του Docker δεν προέρχεται από επιπλέον context switches. Η έλλειψη CPU migrations και οι παρόμοιες τιμές page-faults υποδηλώνουν ότι τα containers δεν εισάγουν σημαντική πρόσθετη επιβάρυνση διαχείρισης μνήμης.

Perf record/report



Η ανάλυση του perf report αποκαλύπτει ότι τα κύρια σημεία συμφόρησης (CPU hotspots) είναι παρόμοια και στα τρία σενάρια (native, layered, volume), αλλά με σημαντικές διαφορές στην κατανομή τους.



Οι κλήσεις συστήματος αποτελούν το μεγαλύτερο hotspot, καταλαμβάνοντας περίπου 33-35% των κύκλων CPU.
Το layered deployment εμφανίζει το υψηλότερο system call overhead (35,0%). Αυτό εξηγεί μεγάλο μέρος της επιπλέον επιβάρυνσης του Docker, καθώς οι μεταβάσεις μεταξύ userspace και kernelspace επιβαρύνονται από τους μηχανισμούς απομόνωσης (namespace isolation) και τους ελέγχους ασφαλείας των containers.
Το volume deployment έχει χαμηλότερο system call overhead (33,5%) από το layered, χάρη στην πιο άμεση πρόσβαση στον δίσκο.

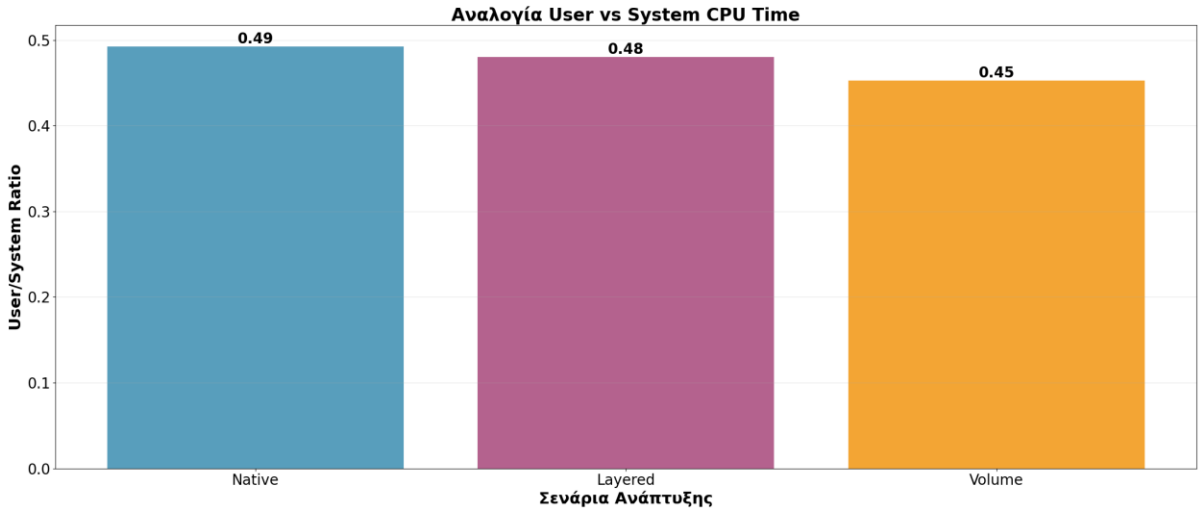
Η μεταγλώττιση Just-In-Time (JIT compilation) είναι έντονη σε όλες τις διαμορφώσεις (10-11% του χρόνου εκτέλεσης).
Το native deployment παρουσιάζει το υψηλότερο JIT overhead (11,07%), πιθανόν επειδή η άμεση πρόσβαση στους πόρους επιτρέπει πιο επιθετική βελτιστοποίηση κώδικα.
Οι containers (Layered/Volume) έχουν οριακά χαμηλότερο JIT overhead, κάτι που, σε συνδυασμό με τις φάσεις Code Generation και Optimization, υποδηλώνει ότι οι μηχανισμοί ασφαλείας και απομόνωσης των containers ενδέχεται να περιορίζουν κάποιες δυνατότητες βελτιστοποίησης.

Η κατανομή αυτών των hotspots εξηγεί τις διαφορές στο συνολικό throughput:
Το native deployment, παρά το υψηλότερο JIT overhead, κερδίζει σε απόδοση χάρη στο χαμηλότερο system call overhead και τις πιο αποτελεσματικές βελτιστοποιήσεις.
Οι υλοποιήσεις σε Container πληρώνουν το τίμημα του overhead που εισάγουν τα στρώματα απομόνωσης και εικονικοποίησης (κυρίως μέσω των system calls), οδηγώντας σε 7-9% χαμηλότερη απόδοση συγκριτικά με το native σενάριο.

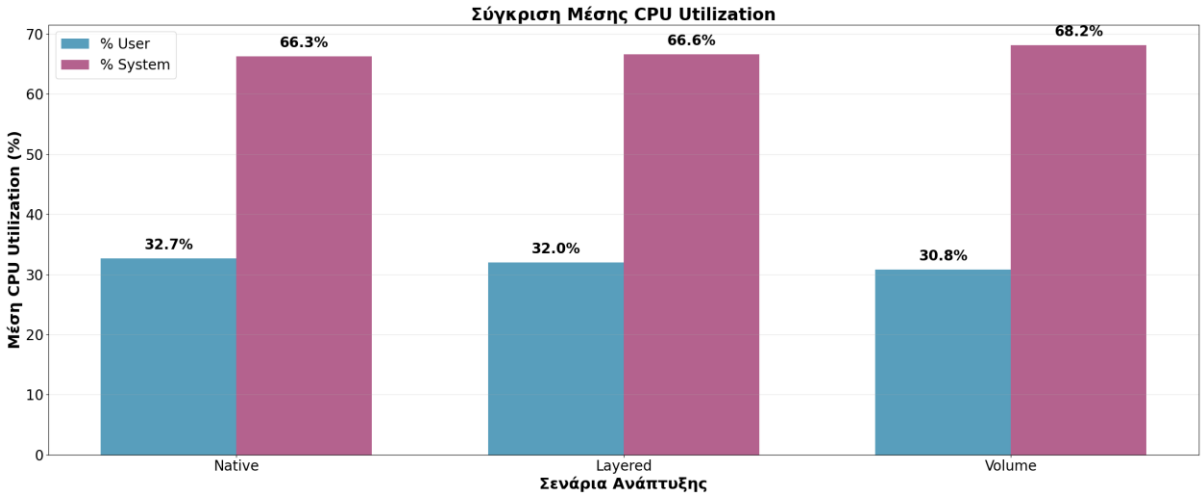
Ανάλυση System Monitoring

Mpstat

Η ανάλυση των δεδομένων mpstat αποκαλύπτει σημαντικές διαφορές στην κατανομή του χρόνου CPU μεταξύ των τριών διαμορφώσεων (native, layered, volume), αν και όλα τα σενάρια λειτουργούν κοντά στο 100% της χωρητικότητας.

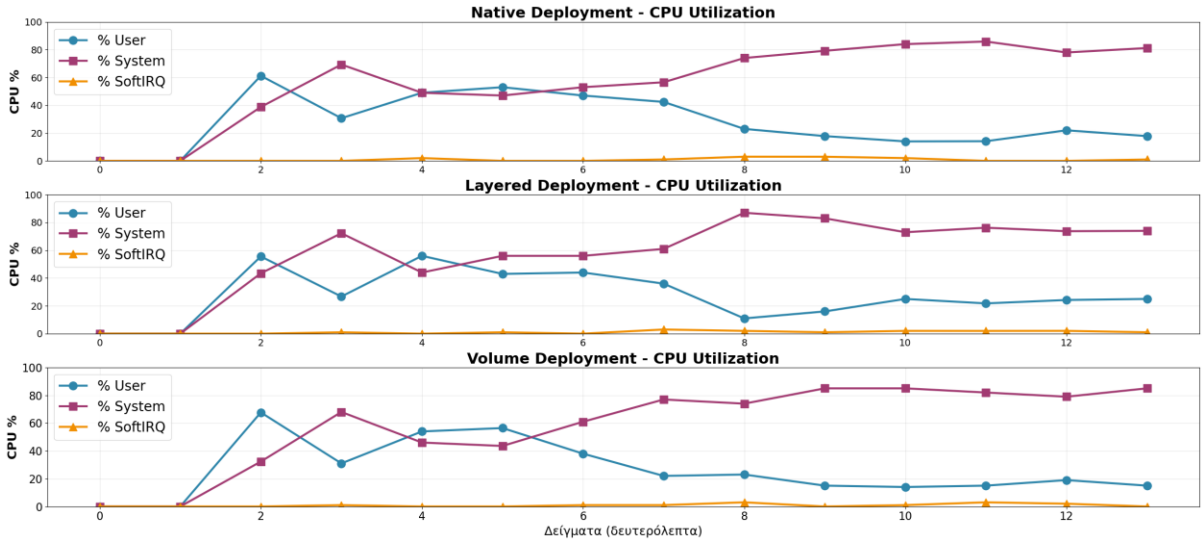


Το κύριο εύρημα είναι η κατανομή του χρόνου μεταξύ εργασίας χρήστη (παραγωγική εργασία) και χρόνου συστήματος (overhead του πυρήνα).
Το native deployment εμφανίζει την υψηλότερη αναλογία user/system time (0.49), υποδηλώνοντας ότι ο χρόνος CPU δαπανάται πιο παραγωγικά, σε πραγματική επεξεργασία δεδομένων και λιγότερο σε εσωτερικές διεργασίες του λειτουργικού συστήματος. Αντίθετα, τα container deployments έχουν χαμηλότερες αναλογίες (Layered: 0.48, Volume: 0.45), υποδεικνύοντας αυξημένο system overhead από το docker runtime.



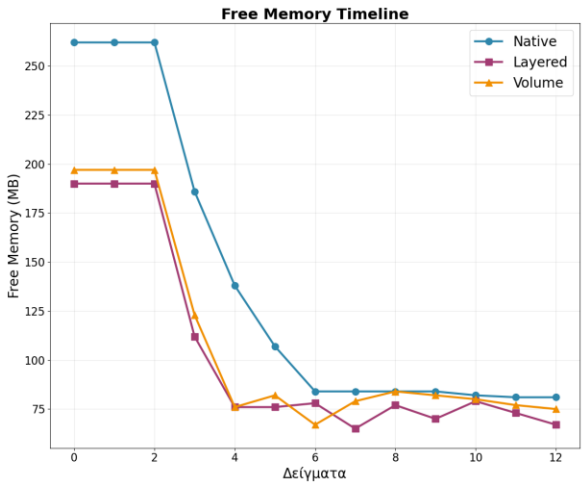
Το layered deployment εμφανίζει μέσο system time 66.6%, ενώ το volume deployment φτάνει ακόμη υψηλότερα levels (68.2%). Αυτή η αύξηση του χρόνου συστήματος στα containers οφείλεται στην πρόσθετη δραστηριότητα του πυρήνα που απαιτείται για τη διαχείριση της απομόνωσης, τους ελέγχους ασφαλείας και την εικονικοποίηση του συστήματος αρχείων. Το native deployment έχει το χαμηλότερο system time (66.3%), επιβεβαιώνοντας το μειωμένο kernel overhead.

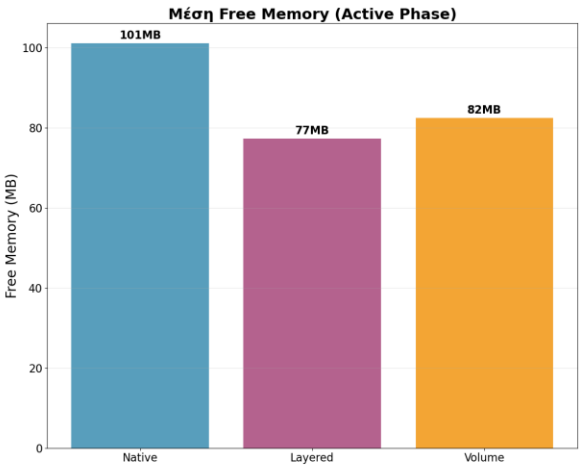
Η κατανομή αυτή εξηγεί άμεσα το γιατί το native deployment, με τον υψηλότερο χρόνο χρήστη (32,7%), επιτυγχάνει και το υψηλότερο throughput. Τα containers, πληρώνουν ένα “πέναλτι” στην απόδοση λόγω του αυξημένου system time.
Επίσης, η σύγκριση μεταξύ των δύο τύπων αποθήκευσης εντός του Docker δείχνει ότι το volume deployment έχει οριακά υψηλότερο system overhead (68,2% έναντι 66,6%), πιθανότατα λόγω της πρόσθετης αφαίρεσης του συστήματος αρχείων που εισάγουν οι τόμοι.



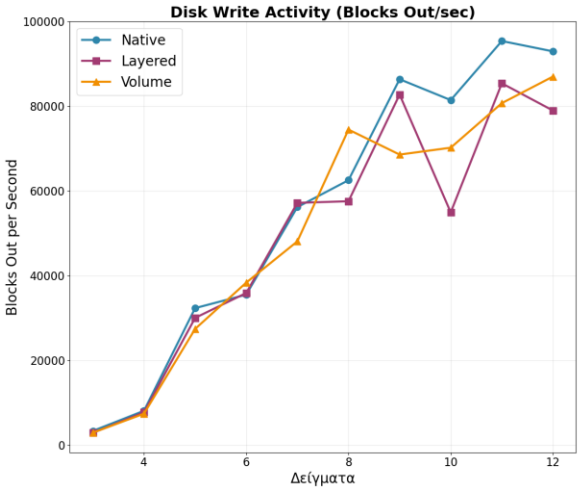
Η δραστηριότητα softIRQ είναι παρόμοια και στα τρία σενάρια (1-3%), υποδηλώνοντας ότι η διαχείριση διακοπών δικτύου δεν αποτελεί σημαντικό bottleneck. Η έλλειψη χρόνου αναμονής I/O (0% σε όλα τα δείγματα) δείχνει ότι το σύστημα δεν δεσμεύεται από I/O, αλλά δεσμεύεται από την CPU.

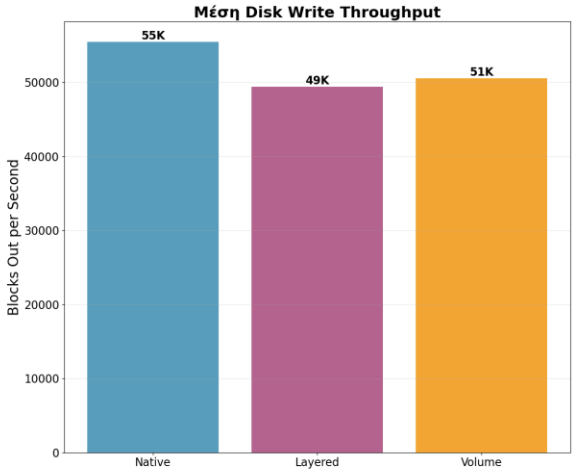
Vmstat



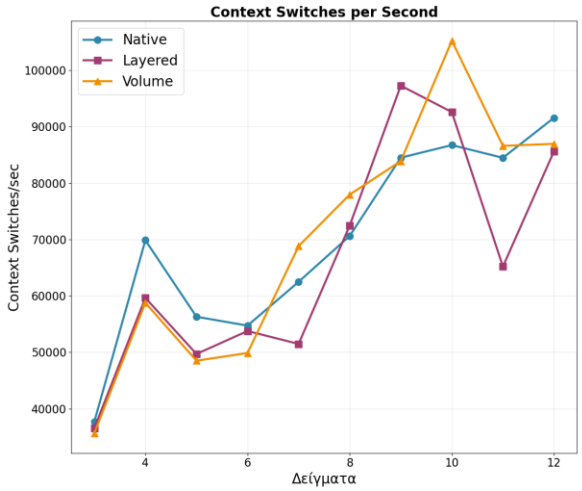


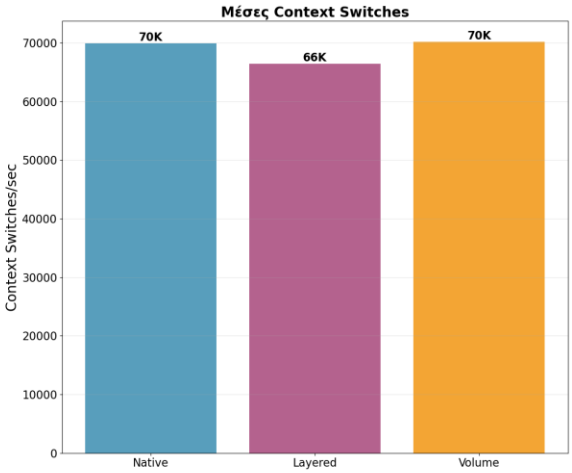
Η ανάλυση αυτή μας δείχνει διαφορές στη χρήση της μνήμης. Το native deployment διατηρεί σημαντικά υψηλότερη μέση ελεύθερη μνήμη (101 MB) σε σχέση με τα containers (Layered: 77 MB, Volume: 82 MB). Αυτό υποδηλώνει ότι τα containers έχουν μεγαλύτερη memory pressure λόγω των απαιτήσεων του Docker runtime και των μηχανισμών απομόνωσης. Επιπλέον, το layered deployment εμφανίζει αξιοσημείωτη δραστηριότητα ανταλλαγής (swap activity), φτάνοντας τα 840 pages/sec, κάτι που οφείλεται στον μηχανισμό copy-on-write του συστήματος αρχείων του Docker.





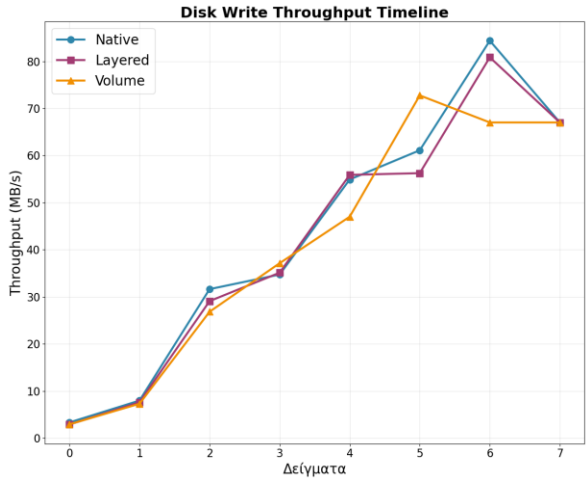
Η δραστηριότητα εγγραφής στο δίσκο (throughput) είναι κρίσιμη για την απόδοση. Το native deployment έχει σημαντικά υψηλότερο μέσο I/O throughput (περίπου 55.000 blocks/sec) σε σύγκριση με τα containers (Layered: ~49.000 blocks/sec, Volume: ~51.000 blocks/sec). Αυτό εξηγεί άμεσα το γιατί το native σύστημα έχει καλύτερη συνολική απόδοση. Εντός των containers, οι τόμοι (volumes) προσφέρουν οριακά καλύτερο I/O performance από το layered filesystem, αλλά με το τίμημα ελαφρώς υψηλότερου system CPU overhead.

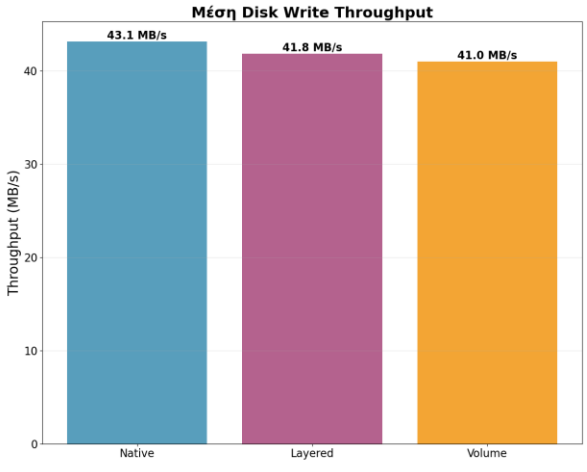




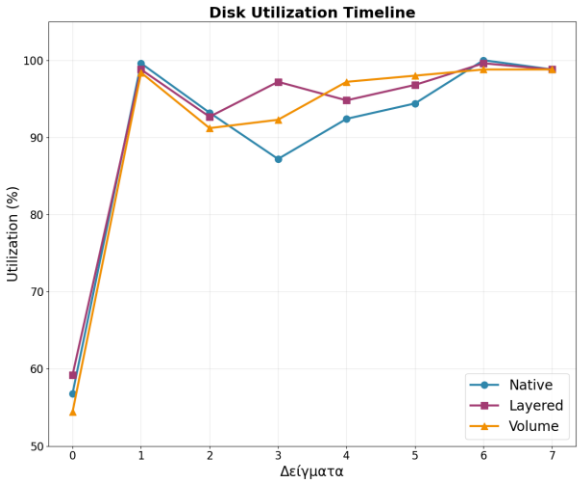
Ο αριθμός των context switches είναι υψηλότερος στο volume deployment (μέση τιμή 70,528 switches/sec) και στο native deployment (69,902 switches/sec), ενώ το layered deployment έχει ελαφρώς χαμηλότερα levels (66,433 switches/sec). Αυτό υποδηλώνει ότι τα native και volume deployments έχουν πιο έντονο task switching, πιθανόν λόγω πιο άμεσης πρόσβασης στους πόρους του συστήματος. Άλλα metrics (run queue length, blocked processes) δείχνουν παρόμοια μοτίβα φόρτου, με το layered deployment να παρουσιάζει περισσότερες διεργασίες σε κατάσταση αναμονής I/O.

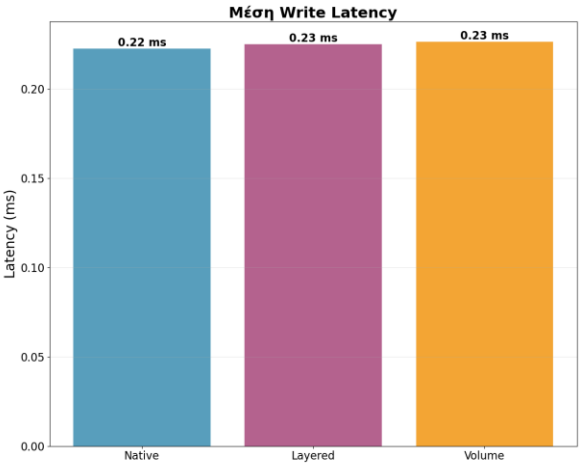
Iostat



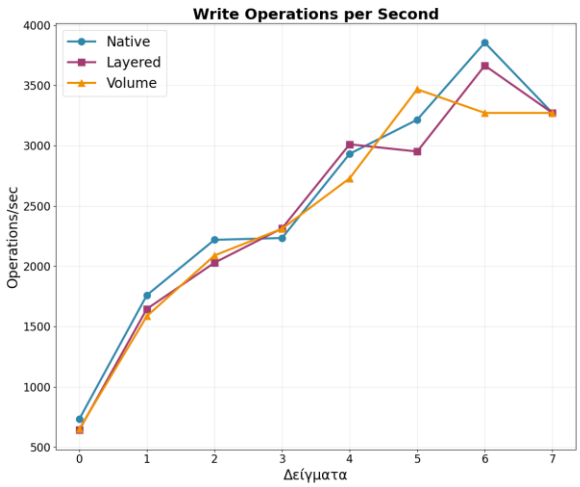


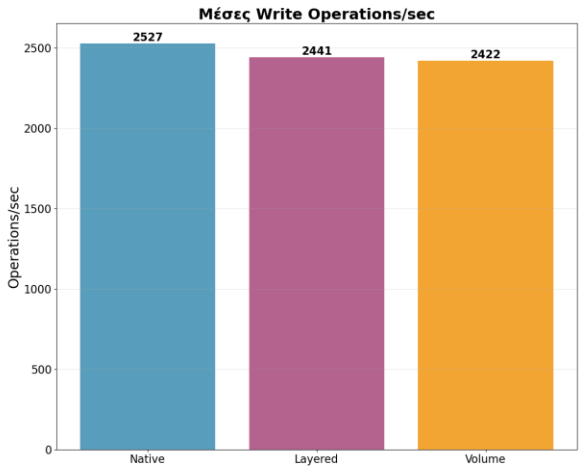
Με τα αποτελέσματα της iostat βλέπουμε ότι το native deployment επιτυγχάνει το υψηλότερο disk write throughput (μέση τιμή 43,1 MB/s), ακολουθούμενο από το layered deployment (41,8 MB/s) και το volume deployment (41,0 MB/s). Αυτή η διαφορά 1,3% μεταξύ native και layered συμβάλλει άμεσα στη συνολική υπεροχή απόδοσης (overall throughput) του native σεναρίου.





Το διάγραμμα της χρησιμοποίησης δίσκου αποκαλύπτει ότι το σύστημα φτάνει σε σημείο κορεσμού. Και στις τρεις περιπτώσεις (Native, Layered, Volume), η χρησιμοποίηση του δίσκου ανεβαίνει γρήγορα πάνω από το 90%, αγγίζοντας συχνά το 100%. Αυτό επιβεβαιώνει ότι ο δίσκος αποτελεί το κύριο σημείο συμφόρησης του συστήματος. Η μέση καθυστέρηση εγγραφής είναι παρόμοια και στα τρία σενάρια (Native: 0,22 ms, Layered: 0,23 ms, Volume: 0,23 ms), υποδηλώνοντας ότι οι drivers των containers δεν προσθέτουν σημαντική επιπλέον καθυστέρηση.





Εδώ παρουσιάζεται ο αριθμός των λειτουργιών εγγραφής ανά δευτερόλεπτο (Write IOPS).Το native σύστημα επιτυγχάνει τον υψηλότερο αριθμό λειτουργιών, κορυφώνοντας στις 2527 writes/sec. Το layered ακολουθεί με 2441 writes/sec, ενώ το volume καταγράφει μέγιστο περίπου 2422 writes/sec. Η συνέπεια αυτής της κατάταξης σε όλες τις μετρήσεις επιβεβαιώνει ότι η απευθείας εκτέλεση native έχει το μικρότερο κόστος (overhead) στις λειτουργίες εισόδου/εξόδου.