ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Persistent B+-Tree Index Evaluation over Heterogeneous DRAM/NVM Systems

Κωνσταντίνα Σ. Σκοβολά

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα
Ιούλιος 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Persistent B+-Tree Index Evaluation over Heterogeneous DRAM/NVM Systems

Κωνσταντίνα Σ. Σκοβολά

Επιβλέπων : Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
....	....	....
Δημήτριος Σούντρης Καθηγητής ΕΜΠ	Σωτήριος Ξύδης Επίκουρος Καθηγητής ΕΜΠ	Δημήτριος Τσουμάκος Αναπληρωτής Καθηγητής ΕΜΠ

Ημερομηνία Εξέτασης:
20 Ιουλίου 2023

Copyright © - All rights reserved Σκοβολά Κωνσταντίνα, 2023.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

(Υπογραφή)

....

Σκοβολά Κωνσταντίνα
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2023 - All rights reserved.

Περίληψη

Η Persistent Memory είναι μια σχετικά νέα κατηγορία συσκευών μνήμης, που βρίσκονται μεταξύ DRAM και Flash στην ιεραρχία της μνήμης, οι οποίες συνδυάζουν τα εξής χαρακτηριστικά: διατηρούν τα δεδομένα τους χωρίς παροχή ρεύματος, ενώ επίσης υποστηρίζουν την άμεση προσπέλαση (τυχαία πρόσβαση) και διευθυνσιοδότηση byte, σε ταχύτητες κοντά σε αυτές της DRAM. Πρόσφατα η Intel διέθεσε στο εμπόριο το πρώτο προϊόν αυτού του τύπου, την Intel® Optane™ Persistent Memory. Ιδιαίτερα οι εφαρμογές βάσεων δεδομένων μπορούν να ωφεληθούν σημαντικά από την έλευση της NVM, καθώς υπάρχει η απαίτηση αφενός να διατηρούν τα δεδομένα τους και να μπορούν να ανακάμπτουν γρήγορα μετά από απώλεια ρεύματος ή αστοχίες του συστήματος, αφετέρου να παρέχουν υψηλή απόδοση και ταχύτητα απόκρισης. Τα τελευταία χρόνια πολλές εργασίες έχουν προτείνει δομές ευρετηρίων (index structures) ειδικά σχεδιασμένες για την Persistent Memory (persistent indexes), οι οποίες έχουν ως στόχο να διατηρήσουν τις υψηλές επιδόσεις, ενώ ταυτόχρονα μπορούν να διατηρούν σημαντικά μεγαλύτερο όγκο δεδομένων σε σχέση με πτητικές δομές ευρετηρίων και καθιστούν το σύστημα ικανό να ανακάμπτει στιγμιαία, καθώς τα δεδομένα τους είναι άμεσα διαθέσιμα και δε χρειάζεται να ανακατασκευαστούν.

Στην παρούσα διπλωματική εργασία γίνεται μια συγκριτική αξιολόγηση της επίδοσης persistent δομών ευρετηρίων βάσεων δεδομένων, χρησιμοποιώντας μια in-memory υλοποίηση του OLTP Benchmark TPCC και το microbenchmark YCSB. Πέρα από την επίδοση, εστιάζουμε επιπλέον και στην κατανάλωση ενέργειας, που είναι μια μετρική την οποία η υπάρχουσα βιβλιογραφία έχει σε μεγάλο βαθμό αγνοήσει. Διαπιστώνουμε ότι η κατανάλωση ενέργειας είναι αντιστρόφως ανάλογη της επίδοσης και ότι διαφορετικές δομές επιτυγχάνουν καλύτερες επιδόσεις ανάλογα με το benchmark και τις σχεδιαστικές επιλογές τους. Για παράδειγμα για το TPCC και 1 warehouse, η καλύτερη επίδοση επιτυγχάνεται από το Fast&Fair και είναι περίπου 53 χιλιάδες συναλλαγές ανά δευτερόλεπτο, για έναν client, επίδοση καλύτερη κατά 8%, 91% και 400% καλύτερη αυτής πουπετυχαίνουν για τις ίδιες παραμέτρους τα WBtree, Masstree, FastFair αντίστοιχα. Για το YCSB, καλύτερη επίδοση για ακέραια κλειδιά έχει το BwTree, ακολουθούμενο στα περισσότερα workloads από το FastFair και το Masstree. Για κλειδιά-συμβολοσειρές, η επίδοση όλων των indexes μειώνεται αρκετά. Τη μικρότερη αρνητική επίδραση παρατηρούμε στην περίπτωση του Masstree, έχοντας χειρότερη επίδοση τουλάχιστον κατά 39%, το οποίο είναι σχεδιασμένο για να χειρίζεται αποτελεσματικά κλειδιά-συμβολοσειρές, ενώ για τα BwTree, FastFair, η επίδοση χειροτερεύει από 3 έως και 6.5 φορές, και για το WBtree, η επίδοση είναι χειρότερη κατά 2 έως 2.5 φορές. Επίσης αυξάνεται η κατανάλωση ενέργειας.

Λέξεις Κλειδιά — B+trees, TPCC, range indexes, databases, indexing, evaluation, Optane, NVM, Persistent Memory, benchmark implementation, heterogeneous DRAM/NVM systems

Abstract

Persistent Memory is an emerging class of memory devices, sitting between DRAM and flash-based storage in the memory hierarchy, and offering data persistence and direct (random) access (byte addressability) at close to DRAM speeds. It recently became commercially available by Intel with the release of its Optane modules. Database applications in particular can greatly profit from NVM, since they are required to provide both durability and speed / high performance. In recent years, many research works have proposed index structures specifically designed for Persistent Memory (persistent indexes), which aim to maintain high performance, while at the same time being able to hold more data compared to volatile indexes and enabling instant recovery through data persistence.

In this thesis we conduct a comparative evaluation of persistent database indexes, using an in-memory implementation of the TPCC OLTP benchmark and the YCSB microbenchmark. In addition to performance, we focus on energy consumption, which is a metric that the existing literature has for the most part ignored. We find that energy consumption is inversely proportional of performance and that different index structures achieve better performance depending on the benchmark/workload and their design choices. For example, for the TPCC benchmark, a single warehouse and 750,000 transactions performed by a single client, the best performing index is FastFair, achieving 53 thousand transactions per second, which is 8%, 91% and 400% better than WBtree, Masstree and BwTree's respective performance. For the YCSB, we find that BwTree performs best, followed by FastFair and then Masstree for most integer workloads. For string workloads we observe a significant performance drop for all index structures. Masstree is the least affected, experiencing at least 39% performance drop. Single-threaded WBtree experiences 2-2.5x performance decrease, FastFair and BwTree performance decreases 3 to 6.5x compared to integer keys, while energy consumption increases.

Keywords — B+trees, TPCC, range indexes, databases, indexing, evaluation, Optane, NVM, Persistent Memory, benchmark implementation, heterogeneous DRAM/NVM systems

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή αυτής της διπλωματικής εργασίας κ. Δημήτριο Σούντρη για την ευκαιρία που μου έδωσε να την εκπονήσω στο εργαστήριο Μικροϋπολογιστών και Ψηφιακών Συστημάτων. Ευχαριστώ επίσης ιδιαίτερα τον υποψήφιο διδάκτορα Μανώλη Κατσαραγάκη για την καθοδήγηση, τη στήριξη και την πολύτιμη βοήθειά του, καθώς και τους μεταδιδακτορικούς ερευνητές και συνεργάτες του εργαστηρίου Χρήστο Μπαλούκα και Λάζαρο Παπαδόπουλο, που μου έδωσαν την ευκαιρία να γνωρίσω και να συμμετάσχω σε μια ακαδημαΙκή ερευνητική ομάδα.

Τέλος, ευχαριστώ την οικογένειά μου που ήταν δίπλα μου στη διάρκεια των σπουδών μου και τους φίλους μου, ειδικά "Το Παραδοσιακό", καθώς και την ομάδα της βιβλιοθήκης ΣΗΜΜΥ που ομόρφυναν τα φοιτητικά μου χρόνια.

Contents

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Contents 11

Εκτενής Ελληνική Περίληψη 13
Εισαγωγή . . . 13
Θεωρητικό Υπόβαθρο . . . 14
Persistent Memory και Intel Optane DCPM . . . 14
Προγραμματισμός Persistent Memory και προκλήσεις . . . 15
Δομές Ευρετηρίων στις Βάσεις Δεδομένων . . . 15
Σχετική Βιβλιογραφία . . . 16
Πειραματική Αξιολόγηση . . . 18
Υλοποίηση TPCC . . . 18
Πειραματική Αξιολόγηση με το TPCC . . . 19
Πειραματική Αξιολόγηση με το YCSB . . . 20
Σύνοψη και Προτεινόμενες Επεκτάσεις . . . 23

1 Introduction 25

2 Theoretical Background 27
2.1 Persistent Memory . . . 27
2.2 Intel Optane DC PMEM . . . 28
2.3 Programming Persistent Memory & PMDK . . . 31
2.3.1 Operating system support for persistent memory . . . 31
2.3.2 Challenges of Programming Persistent Memory . . . 32
2.3.3 Persistent Memory Development Kit (PMDK) . . . 33
2.4 Linearizability . . . 33
2.5 Database Indexes . . . 33
2.5.1 B+-Tree . . . 34
2.5.2 LSM-Tree . . . 37
2.5.3 Skiplist . . . 39
2.5.4 Trie & Radix Tree . . . 39
2.5.5 Hash index . . . 40

3 Literature Review **41**
3.1 Challenges of persistent index structure design . . . 41
3.2 Recent Index Designs for Optane . . . 42
3.2.1 BzTree . . . 42
3.2.2 Masstree . . . 42
3.2.3 PACTree . . . 43
3.2.4 Fast&Fair . . . 43
3.2.5 P-BwTree . . . 43
3.2.6 wBTree . . . 44
3.2.7 DPTree . . . 44
3.2.8 ChameleonDB . . . 45
3.2.9 ViPer . . . 45
3.3 Other evaluation works . . . 46
3.4 Converting DRAM indexes to persistent memory indexes . . . 47

4 Proposed Framework **49**
4.1 TPC-C Benchmark Specification . . . 50
4.2 In-memory TPC-C Benchmark implementation . . . 51
4.2.1 Multiple clients support . . . 52
4.2.2 Mapping of TPCC operations to tree operations . . . 53
4.3 Evaluated Indexes . . . 53

5 Evaluation Results **55**
5.1 Server characteristics and configuration . . . 55
5.2 Evaluation metrics . . . 55
5.3 TPC-C . . . 56
5.3.1 Load phase . . . 56
5.3.2 Run phase . . . 57
5.4 YCSB (microbenchmark) . . . 61
5.4.1 Integer keys . . . 62
5.4.2 String keys . . . 69
5.4.3 Performance comparison of integer and string keys . . . 74

6 Conclusions **77**
6.1 Thesis Summary and Future Work . . . 77

Εκτενής Ελληνική Περίληψη

Εισαγωγή

Η Persistent Memory (PMEM) είναι μια σχετικά νέα τεχνολογία μνήμης, η οποία συνδυάζει τη μη πτητικότητα της μνήμης Flash, με τη διευθυνσιοδότηση σε επίπεδο byte της DRAM. Τεχνολογίες αυτού του τύπου μελετώνται εδώ και αρκετά χρόνια, ωστόσο, μέχρι πολύ πρόσφατα, δεν υπήρχε κάποιο εμπορικά διαθέσιμο προϊόν στην αγορά. Ως εκ τούτου, η έρευνα που αποσκοπούσε στην ενσωμάτωση αυτής της μνήμης σε εφαρμογές, μοντελοποιούσε την PMEM χρησιμοποιώντας DRAM emulation, υποθέτοντας ότι η μόνη διαφορά ήταν πως η PMEM ήταν απλώς μια πιο αργή, μη-πτητική DRAM. Την τελευταία τετραετία περίπου, με την διάθεση της Intel Optane DCPM στην αγορά, κατέστη πλέον δυνατό για τους ερευνητές να αξιολογήσουν τα τεχνικά χαρακτηριστικά ενός πραγματικού προϊόντος, και να σχεδιάσουν εφαρμογές πάνω σε αυτό που να λαμβάνουν υπόψη τα συγκεκριμένα χαρακτηριστικά του. Έτσι φάνηκε ότι οι υποθέσεις του παρελθόντος δεν ευσταθούν, καθώς σε σύκγριση με την DRAM, η PMEM εμφανίζει μεγάλη ασυμμετρία στη συμπεριφορά αναγνώσεων/εγγραφών και έχει μικρότερο bandwidth το οποίο είναι εύκολο να κορεστεί ακόμη και με λίγα νήματα.

Μεταξύ των εφαρμογών που μπορούν να ωφεληθούν ιδιαίτερα από τη χρήση της Persistent Memory, είναι οι εφαρμογές βάσεων δεδομένων, καθώς σκοπός τους είναι να παρέχουν αποδοτικά queries ενώ ταυτόχρονα να εγγυώνται ότι τα δεδομένα τους θα διατηρηθούν σε περίπτωση απώλειας ρεύματος ή αστοχίας του συστήματος. Επιπλέον είναι επιθυμητή η ταχεία ανάκαμψή τους (recovery) σε αυτές τις περιπτώσεις. Ένα από τα σημαντικότερα performance-critical components μιας εφαρμογής βάσεων δεδομένων είναι η δομή ευρετηρίου (index structure) που χρησιμοποιεί, που είναι και πολύ βασικό κομμάτι των NoSQL βάσεων δεδομένων και των key-value stores που γνωρίζουν ιδιαίτερη δημοφιλία τα τελευταία χρόνια. Μέσα σε αυτό το πλαίσιο, πολλά ερευνητικά άρθρα έχουν εξερευνήσει πώς πρέπει να επαναπροσδιοριστεί ο σχεδιασμός τέτοιων δομών, με βάση την υπάρχουσα τεχνολογία και τα πραγματικά της χαρακτηριστικά. Οι ερευνητές έχουν προτείνει νέες δομές ειδικά σχεδιασμένες για την Optane, ωστόσο, δεν είναι τόσο σαφές πώς οι νέες προτάσεις συγκρίνονται μεταξύ τους και με τις προτάσεις του παρελθόντος.

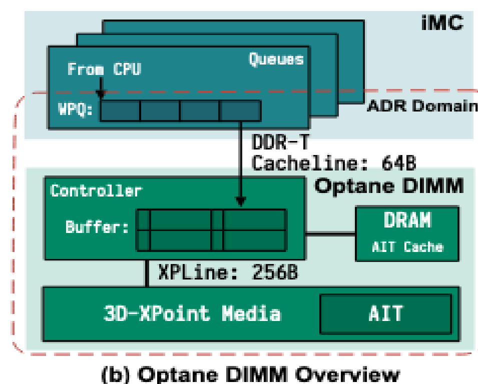
Σε αυτή την εργασία, στόχος μας είναι να προτείνουμε μια μεθοδολογία για την αξιολόγηση της επίδοσης και της κατανάλωσης ενέργειας διαφορετικών δομών ευρετηρίων βάσεων δεδομένων, σχεδιασμένων για ετερογενή συστήματα DRAM/NVM. Η συνεισφορά της αξιολόγησής μας σε σχέση με την υπάρχουσα βιβλιογραφία είναι ότι αφενός λαμβάνουμε υπόψη την κατανάλωση ενέργειας ως μετρική, αφετέρου αξιολογούμε την επίδοση και υπό transactional workloads με το benchmark TPCC, που είναι δύο όψεις που η υπάρχουσα βιβλιογραφία σε μεγάλο μέρος αγνοεί. Έτσι, έχουμε αξιολογήσει τέσσερις δομές αντιπροσωπευτικές διαφορετικών σχεδιαστικών επιλογών καθεμία, χρησιμοποιώντας μια in-memory υλοποίηση του TPCC επιπλέον του standard YCSB, και έχουμε επίσης εστιάσει στην κατανάλωση ενέργειας πέρα από την απόδοση.

Θεωρητικό Υπόβαθρο

Persistent Memory και Intel Optane DCPM

Η Persistent Memory ή αλλιώς Non-Volatile Memory (NVM) είτε Storage Class Memory (SCM) είναι μια σχετικά νέα τεχνολογία μη-πτητικής μνήμης. Υπάρχουν διάφορες τεχνολογίες που υπάγονται σε αυτήν την κατηγορία, όπως Phase Change Memory (PCM), Spin-Transfer Torque RAM(STT-RAM), Resistive RAM (Re-RAM) και 3D-XPoint. Τα βασικά χαρακτηριστικά της είναι ότι είναι byte-addressable, όπως και η DRAM, δηλαδή μπορεί και αυτή να προσπελαστεί άμεσα από τον επεξεργαστή με εντολές load/store. Έχει επίσης υψηλή πυκνότητα και χαμηλό κόστος ανά bit, χαμηλότερη κατανάλωση ενέργειας και μικρότερη αντοχή από ό,τι η DRAM. Ως προς το access latency, αυτό είναι στην ίδια τάξη μεγέθους με της DRAM, ωστόσο σημαντικά μεγαλύτερο, ειδικά όσον αφορά τις εγγραφές (writes), καθώς ένα ακόμη αξιοσημείωτο χαρακτηριστικό είναι η ασυμμετρία μεταξύ της καθυστέρησης των αναγνώσεων και των εγγραφών: σε αντίθεση με την DRAM, οι αναγνώσεις είναι τουλάχιστον 3 φορές πιο γρήγορες από τις εγγραφές στην Optane.

Η πρώτη εμπορικά διαθέσιμη μνήμη αυτού του τύπου είναι η Optane DCPM που διατίθεται από την Intel, η οποία είναι βασισμένη στην τεχνολογία 3D-XPoint και είναι συμβατή με επεξεργαστές Cascade Lake και νεότερους. Η μνήμη αυτή τοποθετείται στο memory bus και συνδέεται με τον επεξεργαστή με τον ίδιο τρόπο που συνδέονται και τα DIMMS της DRAM και έχει δύο τρόπους λειτουργίας: μπορεί είτε να λειτουργήσει ως επέκταση της DRAM (Memory Mode) στην οποία περίπτωση ουσιαστικά συνιστά μια μεγαλύτερης χωρητικότητας πτητική μνήμη, είτε να λειτουργήσει στο Application-Direct Mode που είναι και ο τρόπος λειτουργίας που εκμεταλλεύεται τη μη-πτητικότητα της μνήμης. Στο πλαίσιο αυτής της διπλωματικής, δε μας ενδιαφέρει το Memory Mode. Σημαντικό να σημειωθεί ότι ενώ, ο memory controller επικοινωνεί με την NVM βάσει του πρωτοκόλλου DDR-T χρησιμοποιώντας μπλοκς των 64 bytes, το μέγεθος σελίδας της Optane είναι τα 256 Bytes. Συνεπώς, loads και stores μικρότερα των 256B απλώς σπαταλούν το διαθέσιμο bandwidth και προκαλούν amplification. Ο τομέας που ονομάζεται ADR (Asynchronous DRAM Refresh) είναι ο τομέας που, αν τα δεδομένα φτάσουν σε αυτόν, είναι εγγυημένη η μη-πτητικότητά τους. Κομμάτι του είναι το WPQ (write-pending queue), όπου τοποθετούνται οι εγγραφές αφού εκτελεστεί ένα flush instruction, και φυσικά η ίδια η Persistent Memory. Τα παραπάνω, καθώς και ο τρόπος διασύνδεσης ενός Optane DIMM απεικονίζονται σχηματικά στην Εικόνα 1.



Εικόνα 1: Optane DIMM Overview

Προγραμματισμός Persistent Memory και προκλήσεις

Προκειμένου τα δεδομένα να φτάσουν από την πτητική κρυφή μνήμη της CPU στην Optane, και να θεωρηθούν πλέον μη πτητικά, ο προγραμματιστής ζητά ρητά την εγγραφή τους. Για το σκοπό αυτό χρησιμοποιούνται οι εντολές clflush, clflushopt, clwb οι οποίες γράφουν στη μνήμη μια γραμμή της cache, mfence/sfence οι οποίες είναι memory fences που ο σκοπός τους είναι να σειριοποιούν τις εγγραφές, εξασφαλίζοντας ότι θα γίνουν με τη σειρά που ορίζει ο προγραμματιστής, και temporal stores (movnt) οι οποίες αποθηκεύουν τις εγγραφές σε έναν ειδικό buffer, τα περιεχόμενα του οποίου γράφονται στη μνήμη όταν γεμίσει είτε όταν ρητά εκτελεστεί μια εντολή sfence.

Ο προγραμματισμός Persistent Memory δεν είναι εύκολη διαδικασία, καθώς παρουσιάζει επιπλέον προκλήσεις σε σχέση με τον παραδοσιακό προγραμματισμό σε DRAM. Οι βασικότερες από αυτές είναι οι ακόλουθες:

Συνέπεια Δεδομένων. Η πρόκληση αυτή αφορά στην ανάγκη τα δεδομένα να εγγράφονται στην PMEM με την σωστή σειρά, παρά το γεγονός ότι οποιεσδήποτε γραμμές της cache μπορεί να αδειάσουν ανά πάσα στιγμή. Η σωστή σειρά των εγγραφών διασφαλίζεται με τη χρήση εντολών clflush/sfence.

Ανάκτηση Δεδομένων. Όταν επανεκκινεί ένα πρόγραμμα, για παράδειγμα μετά από ένα κρασάρισμα, ο χώρος εικονικών διευθύνσεων που του δίνεται μπορεί να διαφέρει από την προηγούμενη εκτέλεση. Αυτό αποτελεί μια δυσκολία για το πώς αποθηκεύονται οι δείκτες(pointers) στην PMEM. Η λύση συνήθως είναι να αποθηκεύεται ένα offset, αντί για συγκεκριμένη διεύθυνση, το οποίο μετά προστίθεται στη νέα διεύθυνση για την ανάκτηση των δεδομένων.

Διαρροές Persistent Memory. Υπάρχουν δύο ειδών διαρροές: Πρώτον, αυτές που υπάρχουν και στον παραδοσιακό προγραμματισμό, όταν μια εφαρμογή αμελεί να απελευθερώσει μνήμη που δέσμευσε δυναμικά, με τη διαφορά ότι μετά την εκτέλεση του προγράμματος, η μνήμη αυτή παραμένει δεσμευμένη καθώς διατηρούνται τα περιεχόμενα της NVM. Επιπλέον υπάρχει κι ένα δεύτερο είδος διαρροών, που μπορούν να συμβούν κατά το memory allocation. Αν για παράδειγμα κρασάρει το σύστημα ενώ γίνεται ένα allocation, υπάρχει περίπτωση ο allocator να βλέπει πως ένα κομμάτι της μνήμης έχει διατεθεί στην εφαρμογή, ενώ η εφαρμογή να μην μπορεί να το δει.

Ημιτελείς εγγραφές Οι σύγχρονες CPUs υποστηρίζουν ατομικές εγγραφές μέχρι 8 bytes. Ωστόσο είναι σύνηθες το μέγεθος μιας εγγραφής να υπερβαίνει τα 8 bytes. Έτσι μια εφαρμογή για να κάνει 'ατομικές' αλλαγές μεγάλου μεγέθους, πρέπει να εκμεταλλευτεί τις ατομικές λειτουργίες που υποστηρίζει το υλικό, και συνήθως αυτό γίνεται χρησιμοποιώντας μικρότερα πεδία μεγέθους έως 8 bytes, που μπορούν να ενημερωθούν ατομικά και δείχνουν την κατάσταση μιας μεγαλύτερης εγγραφής.

Δομές Ευρετηρίων στις Βάσεις Δεδομένων

Ένα σημαντικό τμήμα της έρευνας στον τομέα της αξιοποίησης της Persistent Memory σε εφαρμογές αφορά στις βάσεις δεδομένων και πιο ειδικά, στις δομές ευρετηρίων τους. Τα ευρετήρια έχουν θεμελιώδη σημασία για τη βελτίωση της απόδοσης της βάσης δεδομένων και τη γρήγορη ανάκτηση δεδομένων. Κάποιες από τις κύριες μορφές ευρετηρίων είναι τα B+-trees, LSM-trees, Tries και Radix trees, οι πίνακες κατακερματισμού(hashtables), οι skiplists. Ορισμένες από τις δομές αυτές είναι παραδοσιακά σχεδιασμένες για την DRAM. Οι δομές αυτές περιγράφονται αναλυτικά στην αντίστοιχη ενότητα του Chapter 2. Εδώ επιγραμματικά αναφέρουμε:

- B+-δέντρα. Τα B+-δέντρα είναι ισορροπημένες δενδρικές δομές ευρετηρίου που χρησιμοποιούνται ευρέως στα συστήματα βάσεων δεδομένων. Οι εσωτερικοί κόμβοι του δέν-

τρου δεν φυλάσσουν δεδομένα, μόνο δείκτες προς τα κατώτερα επίπεδα του δέντρου, ενώ τα δεδομένα βρίσκονται στους κόμβους-φύλλα. Είναι ιδιαίτερα αποτελεσματικές δομές για αναζητήσεις και αναζητήσεις εύρους αλλά λιγότερο για εγγραφές/διαγραφές, καθώς αυτές οι λειτουργίες μπορεί να επιφέρουν επιπρόσθετες δομικές αλλαγές στο δέντρο προκειμένου να διατηρήσει την ισορροπημένη δομή του.

- LSM-δέντρα. Τα LSM(Log Structured Merge) δέντρα εμφανίστηκαν ως εναλλακτική των B+-δέντρων για την υποστήριξη ταχύτερων εγγραφών. Είναι πολυεπίπεδες δομές, όπου το πρώτο επίπεδο είναι πτητικό και τα επόμενα μη-πτητικά. Κάθε επίπεδο μπορεί να έχει διαφορετική υλοποίηση. Στο αρχικό άρθρο που πρότεινε τη δομή αυτή, κάθε επίπεδο ήταν υλοποιημένο ως ένα B+-δέντρο. Οι νέες εγγραφές εισάγονται στο πρώτο επίπεδο, το οποίο όταν γεμίσει, ταξινομείται και τα ταξινομημένα περιεχόμενα συγχωνεύονται με το επόμενο επίπεδο. Η ίδια διαδικασία ταξινομημένης συγχώνευσης εφαρμόζεται και για τα επόμενα επίπεδα, πλην του τελευταίου, όταν φτάνουν ένα συγκεκριμένο μέγεθος.

- Tries και Radix trees. Είναι και αυτές δενδρικές δομές, σχεδιασμένες για τη φύλαξη κλειδιών-συμβολοσειρών, βελτιστοποιημένες για αποδοτική αναζήτηση. Σε ένα Trie κάθε κόμβος φυλάσσει ένα χαρακτήρα, ενώ σε ένα Radix tree μπορεί να φυλάσσει ένα πρόθεμα που αποτελείται από πολλούς χαρακτήρες.

- Πίνακες κατακερματισμού. Οι πίνακες κατακερματισμού είναι δομές σταθερού μεγέθους, που υποστηρίζουν πολύ αποτελεσματικές αναζητήσεις για ένα στοιχείο, αλλά δεν υποστηρίζουν αναζητήσεις σε εύρος.

- Skiplists. Οι skiplists είναι πιθανοτικές δομές που αποτελούν μια εναλλακτική έναντι των B+-δέντρων, καθώς υποστηρίζουν τις ίδιες λειτουργίες με αυτά, αλλά δεν έχουν την ίδια ανάγκη για δομικές αλλαγές προκειμένου να διατηρήσουν αυστηρή ισορροπημένη δομή.

Σχετική Βιβλιογραφία

Πολλές δομές ευρετηρίων έχουν προταθεί από ερευνητές. Γενικοί στόχοι στη σχεδίαση μιας persistent-memory δομής ευρετηρίου είναι: να κρατηθεί ο αριθμός των persist-operations(clwb, clflush, sfence) χαμηλός κατά το δυνατόν, καθώς οι προσβάσεις στην Optane αφενός είναι ακριβές, αφετέρου οι εγγραφές προκαλούν φθορά της μνήμης. Επίσης, καθώς χρειάζεται να αποφεύγονται περιττές εγγραφές, γίνεται προσπάθεια να μειωθούν οι λειτουργίες για τη διατήρηση μεταδεδομένων και δομικών ιδιοτήτων. Ένας ακόμη στόχος είναι η δομή να υποστηρίζει ταυτόχρονες λειτουργίες ακόμη και για πολλά νήματα, ενώ να μη σπαταλά το bandwidth το οποίο είναι περιορισμένος πόρος για την Persistent Memory.

Γενικά σκοπός είναι να επιτευχθεί καλή επίδοση, αποτελεσματικός συγχρονισμός που επιτρέπει την κλιμάκωση για πολλά νήματα, και ταυτόχρονα να διατηρηθούν τα δεδομένα σε περίπτωση απώλειας ρεύματος ή αστοχίας του συστήματος για γρήγορη ανάκαμψη του συστήματος. Οι ερευνητές έχουν αναγνωρίσει κάποιες κοινές 'βασικές' τεχνικές:

Για τη βελτίωση της επίδοσης είναι σύνηθες να εφαρμόζεται "selective persistence", δηλαδή οι καταχωρήσεις στην Optane να περιορίζονται στις απαραίτητες για την ανάκαμψη, και κατά τα άλλα, να χρησιμοποιείται η DRAM ως ταχύτερη. Για παράδειγμα στην περίπτωση των B+-δέντρων, που φυλάσσουν δεδομένα μόνο στα φύλλα, αυτό μπορεί διαισθητικά να εφαρμοστεί τοποθετώντας τα φύλλα μόνο στην Persistent Memory και τα προηγούμενα επίπεδα του δέντρου (που μπορούν

να ανακατασκευαστούν βάσει των φύλλων) στη DRAM. Άλλες διαδεδομένες τεχνικές είναι η χρήση αταξινόμητων κόμβων (unsorted nodes) στην PMEM, ενώ διατηρείται μια ακόμη δομή, συνήθως ένας πίανακας (indirection slot array) στην DRAM που κρατάει την πληροφορία για την ταξινομημένη σειρά των στοιχείων, καθώς και το fingerprinting. Το fingerprint, είναι ένα 1-byte hash για κάθε κλειδί που υπάρχει στο φύλλο. Τα fingerprints τοποθετούνται στο πρώτο cacheline-sized τμήμα του κόμβου-φύλλου, επιταχύνοντας την αναζήτηση καθώς αποφαίνονται για την ύπαρξη ή μη ενός κλειδιού χωρίς να χρειαστεί η διάσχιση του φύλλου. Για την κλιμάκωση, προτιμώνται οι λύσεις που αποφεύγουν τα πολλά/βαριά κλειδώματα, όπως lock-free/optimistic locking. Παρόλα αυτά, οι σχεδιαστικές επιλογές μπορεί να ποικίλλουν. Στον Πίνακα 1 συνοψίζονται οι σχεδιαστικές επιλογές ορισμένων αντιπροσωπευτικών προτάσεων των τελευταίων ετών.

Δομή	Αρχιτεκτονική	Δομή κόμβων	Συγχρονισμός	Κλειδιά Συμβολοσειρές
wBTree	B+-tree;PMEM-only	Unsorted, indirection slot array	Single-threaded	Pointer to key
FPTree	B+-tree; selective persistence; inner nodes in DRAM, leaf nodes in PMEM	Unsorted leaf nodes; fingerprints	HTM & locking	Pointer
BzTree	B+-tree;PMEM-only	Partially unsorted leaf; sorted inner nodes	lock-free(PMwCAS)	Inline
DPTree	selective persistence; B+-tree and inner trie in DRAM; trie leaf in PMEM	unsorted leaf; fingerprints; indirection; extra metadata	optimistic locking; async updates	Pointer
PACTree	Trie;PMEM-only (optional selective persistence)	Unsorted leaf; fingerprints; indirection	optimistic locking; async updates	Inline
FastFair	B+-tree;PMEM-only	Sorted nodes	Lock-free reads; blocking writes	Pointer
Masstree	Hybrid: trie-like concatenation of B+-trees;converted	unsorted leaf; sorted internal;	lock-free reads;write exclusion	Inline
BwTree	B+-tree;converted	logical pages; mapping table;deltas prepended to node;	non-blocking reads and writes	Inline

Πίνακας 1: Σχεδιαστικές Επιλογές ανά δομή

Άλλα άρθρα που κάνουν μια συστηματική σύγκριση και αξιολόγηση persistent indexes είναι οι εργασίες των Lersch et al.[1] καθώς επίσης και He et al.[2], όπου οι ερευνητές συγκρίνουν μεταξύ τους range indexes, δηλαδή δομές ευρετηρίων που μπορούν να υποστηρίξουν ερωτήματα εύρους, όπως είναι τα B+-δέντρα. Μια ακόμη παρεμφερής μελέτη είναι των Hu et al.[3], οι οποίοι αξιολόγησαν την επίδοση δομών κατακερματισμού στηριζόμενοι στην εργασία των Lersch et alγια το benchmarking framework. Τα ευρετήρια που συμπεριέλαβαν στις αξιολογήσεις είναι αντιπροσωπευτικά διαφορετικών σχεδιαστικών επιλογών και concurrency schemes.

Παράλληλα με την πρόταση νέων δομών δεδομένων ειδικά σχεδιασμένων για την Optane, που λαμβάνουν υπόψη τα ιδιαίτερα χαρακτηριστικά της σε σχέση με την DRAM ώστε να βελτιστοποιήσουν την επίδοσή τους ενώ ταυτόχρονα διατηρούν τα δεδομένα τους σε περίπτωση απώλειας ρεύματος, έχουν προταθεί μεθοδολογίες για τη μετατροπή ήδη υπαρχουσών πτητικών δομών σε μη πτητικές και την κατάλληλη προσαρμογή τους στην PMEM. Άρθρα όπως το RECIPE[4], NVTraverse[5], PRONTO[6] και το TIPS[7], προτείνουν τεχνικές μετασχηματισμού.

Ειδικότερα το RECIPE, δεν παρέχει προγραμματιστικές διεπαφές αλλά παρέχει κανόνες για την προσαρμογή του πηγαίου κώδικα για τη μετατροπή. Διακρίνει τρεις κατηγορίες πτητικών δομών που μπορούν να προσαρμοστούν για την Persistent Memory, ανάλογα με τα κλειδώματα που παίρνουν οι αναγνώσεις/εγγραφές, concurrency scheme που χρησιμοποιείται και τον τρόπο που γίνονται οι ενημερώσεις. Οι οδηγίες για τη μετατροπή κάθε κατηγορίας, ουσιαστικά διευκρινίζουν στον προγραμματιστή σε ποια σημεία του κώδικα οφείλει να τοποθετήσει εντολές cache line flush και memory fences. Σε αυτή την εργασία, δύο από τις δομές της αξιολόγησης έχουν προσαρμοστεί βάσει του RECIPE για την Persistent Memory.

Το NVTraverse παρομοίως με το RECIPE, καθορίζει τα σημεία στον κώδικα όπου πρέπει να προστεθούν cache line flushes/memory fences, για μια νέα κλάση δομών που ορίζει ως traversal

data structures. Το Pronto χρησιμοποιεί μια τεχνική που ονομάζεται Asynchronous Semantic Logging για να μετατρέψει σε μη πτητικές τις πτητικές δομές, διατηρώντας logs, ενώ το TIPS, επίσης συνδυάζει UNDO και Operational logging αλλά εισάγει προγραμματιστικές διεπαφές που δεν απαιτούν από τον προγραμματιστή να χρησιμοποιήσει flushes και fences.

Όλες οι παραπάνω εργασίες αξιολογούν την αποτελεσματικότητα των μετατροπών που προτείνουν, συγκρίνοντάς τες με state-of-the-art δομές ειδικά σχεδιασμένες για την Persistent Memory. Οι αξιολογήσεις τους δείχνουν ότι, η επίδοση που επιτυγχάνουν οι μετασχηματισμένες δομές, είναι ανάλογη ή καλύτερη αυτής των ειδικά σχεδιασμένων για την Persistent Memory δομών ευρετηρίων. Επιπλέον, η έρευνα των He et al. είχε ένα ενδιαφέρον ανάλογο εύρημα, από την αντίθετη κατεύθυνση: όταν εκτέλεσαν benchmarks στην DRAM με δομές ειδικά σχεδιασμένες για την Persistent Memory, αλλά αφαιρώντας τις εντολές cache line flushes και memory fences, διαπίστωσαν ότι η επίδοση είναι παρεμφερής, σε ορισμένες περιπτώσεις και καλύτερη, μιας δομής βελτιστοποιημένης για την DRAM. Αυτά τα αποτελέσματα είναι ενθαρρυντικά για μια δυνατή ενοποίηση στο σχεδιασμό δομών ευρετηρίων, χωρίς δηλαδή να απαιτείται διαφοροποίηση των σχεδιαστικών επιλογών ανάλογα με τον τύπο μνήμης για τον οποίο προορίζονται.

Πειραματική Αξιολόγηση

Για την αξιολόγηση χρησιμοποιήσαμε δύο benchmarks, το TPC-C και το YCSB. Οι δομές που συμπεριλήφθηκαν στην αξιολόγηση ήταν οι P-Masstree, P-BwTree (οι μη-πτητικές εκδοχές των Masstree, BwTree αντίστοιχα, προσαρμοσμένων για την PMEM από τους συγγραφείς του RECIPE), Fast&Fair και WBtree.

Υλοποίηση TPCC

Το benchmark TPC-C αποτελεί ένα industry standard benchmark για την αξιολόγηση της επίδοσης OLTP (OnLine Transactional Processing) συστημάτων βάσεων δεδομένων. Ο στόχος του είναι να προσομοιώσει ένα περιβάλλον επιχείρησης που περιλαμβάνει διάφορους τύπους συναλλαγών, όπως νέες παραγγελίες πελατών, πληρωμές, έλεγχο αποθεμάτων και κατάστασης παραγγελιών. Για το πειραματικό μέρος της διπλωματικής αυτής, έχουμε βασιστεί σε μια in-memory υλοποίηση του benchmark[8] αυτού, την οποία έχουμε επεκτείνει για να χρησιμοποιήσουμε το TPCC στην αξιολόγηση δομών ευρετηρίων σχεδιασμένων για την PMEM. Η υλοποίηση έχει γίνει σε C++ και το διάγραμμα UML φαίνεται στην Εικόνα 4.3 του Κεφαλαίου 4. Κάθε δομή που αξιολογείται με το TPCC, πρέπει να υλοποιεί τις συναρτήσεις του API της διεπαφής που φαίνεται στο Listing 1 μέσω μιας κλάσης wrapper, η οποία γίνεται compile σαν shared library που φορτώνεται δυναμικά κατά την εκτέλεση του προγράμματος.

```
1  class TreeApi;
2  extern "C" TreeApi *create_tree(tree_options_t *opt);
3
4  class TreeApi
5  {
6  public:
7      virtual ~TreeApi(){};
8      virtual void insert(const void *Nkey, const void *Nvalue) {};
9      virtual bool find(const void *key, void *value = nullptr) const { return 0; };
10     virtual bool remove(const void *key) {return 0;};
11     virtual bool findLastLessThan(const void *key, void *value = nullptr, void *
       out_key = nullptr) const {return 0;};
```

```
12 };
```

Listing 1: TPC-C API

Για να υποστηρίξουμε πολλαπλούς clients σε ένα transactional περιβάλλον, όπως απαιτεί το standard του TPCC, υλοποιούμε το επίπεδο απομόνωσης σειριοποιησιμότητας (serializability). Για την υλοποίηση της σειριοποιησιμότητας, εφαρμόζουμε strict two-phase locking (2PL). Κάθε συναλλαγή που πρόκειται να γράψει σε μία σχέση, παίρνει ένα κλείδωμα εγγραφής σε ολόκληρη τη σχέση και αντίστοιχα κάθε συναλλαγή που πρόκειται να διαβάσει, παίρνει ένα κλείδωμα ανάγνωσης, τα οποία κρατάει μέχρι το τέλος της. Τα κλειδώματα έχουν υλοποιηθεί χρησιμοποιώντας την κλάση της C++ `std::shared_mutex`.

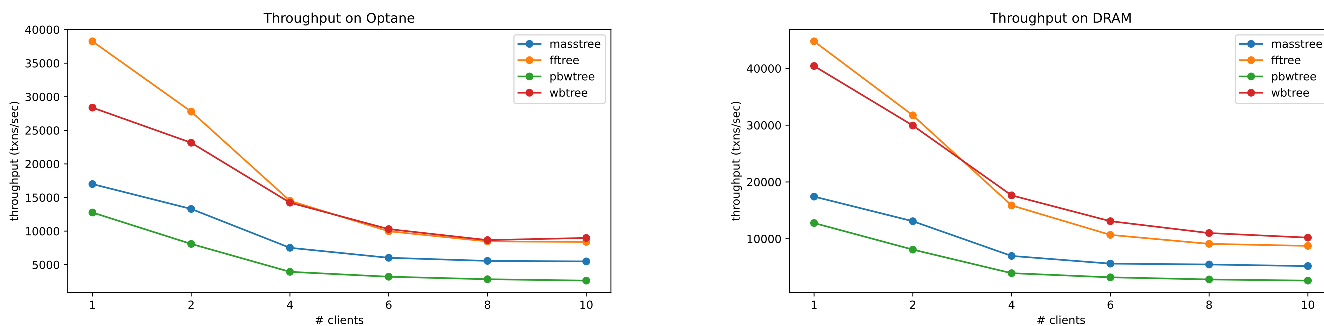
Τα πειράματα έγιναν σε server με 128GiB DRAM, 1536 GiB Optane μοιρασμένα σε 6x256 GiB DIMMs.

Πειραματική Αξιολόγηση με το TPCC

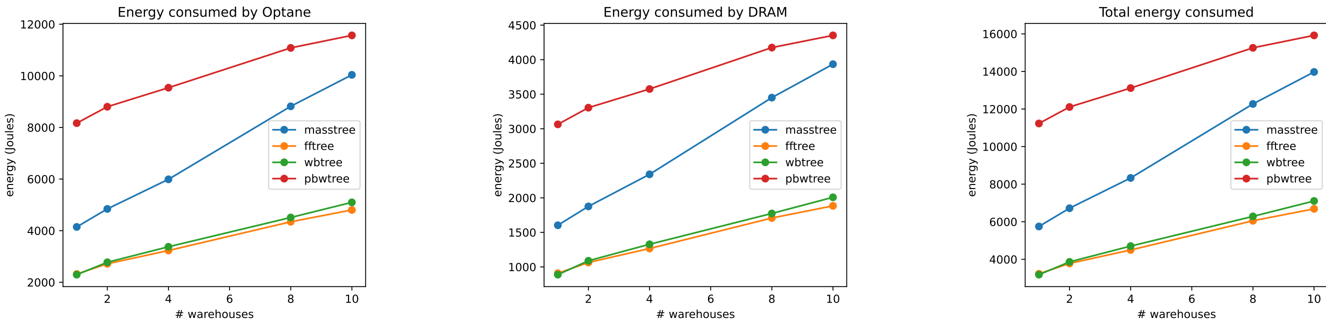
Στις Εικόνες 2 έως 5 παρουσιάζεται η απόδοση (συναλλαγές ανά δευτερόλεπτο), η κατανάλωση ενέργειας, και οι εγγραφές και αναγνώσεις στην Optane κατά την εκτέλεση του προγράμματος. Έχουμε μετρήσει τις διαφορές στην απόδοση για εκτέλεση σε Optane και σε DRAM. Παρατηρήσαμε ότι η επίδοση είναι ελαφρώς μόνο καλύτερη για όλα τα indexes για εκτέλεση στην DRAM. Παρατηρούμε ότι την καλύτερη επίδοση σημειώνει το Fast&Fair, αλλά και το WBtree. Επίσης παρουσιάζουν παρόμοια κατανάλωση ενέργειας.

Ως προς την κατανάλωση ενέργειας, παρατηρούμε ότι είναι αντιστρόφως ανάλογη της επίδοσης, και ότι όλες οι δομές έχουν την ίδια συμπεριφορά και στους δύο τύπους μνήμης: αυτές που καταναλώνουν περισσότερη ενέργεια στην Optane καταναλώνουν και περισσότερη ενέργεια στην DRAM.

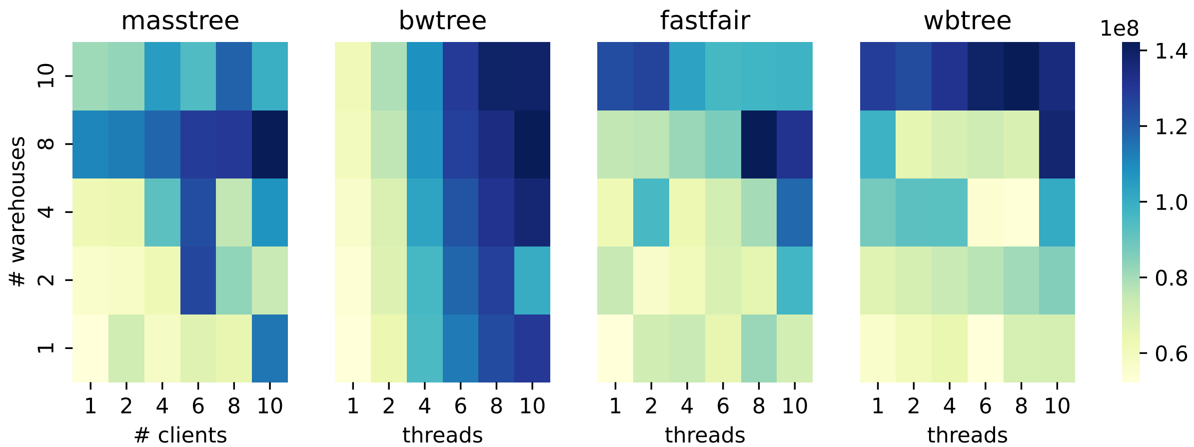
Ως προς τις προσβάσεις στη μνήμη, τα δέντρα με την καλύτερη επίδοση φαίνεται να έχουν παρόμοια συμπεριφορά. Επίσης, καθώς αυξάνεται το πλήθος των clients και ο αριθμός των warehouses, και μειώνεται το throughput, αυξάνεται και το πλήθος των προσβάσεων στη μνήμη.



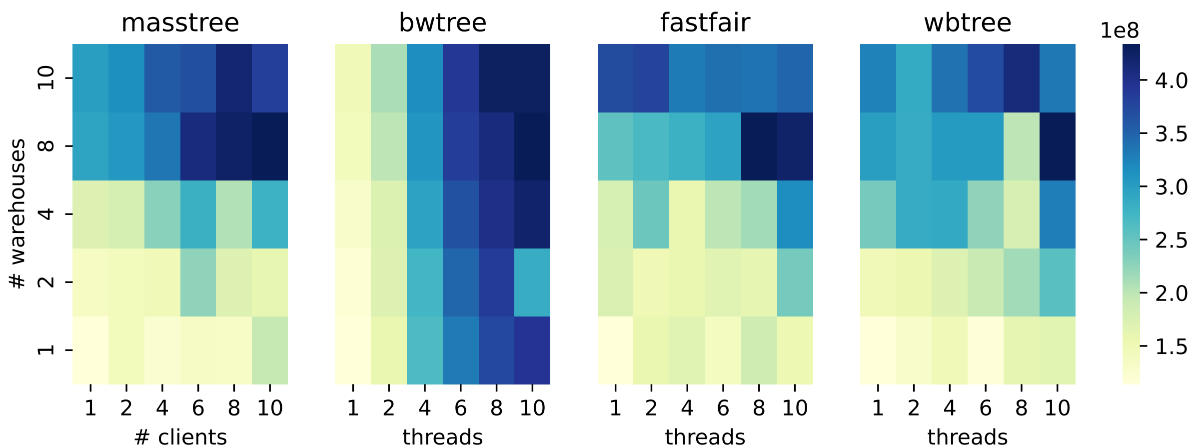
Εικόνα 2: Comparative throughput on Optane and DRAM, 10 warehouses, 750K transactions, increasing number of clients



Εικόνα 3: Consumed energy when running on Optane, 10 warehouses, 750K transactions, increasing number of clients



Εικόνα 4: TPCC Optane write accesses, 750K transactions



Εικόνα 5: TPCC Optane read accesses, 750K transactions

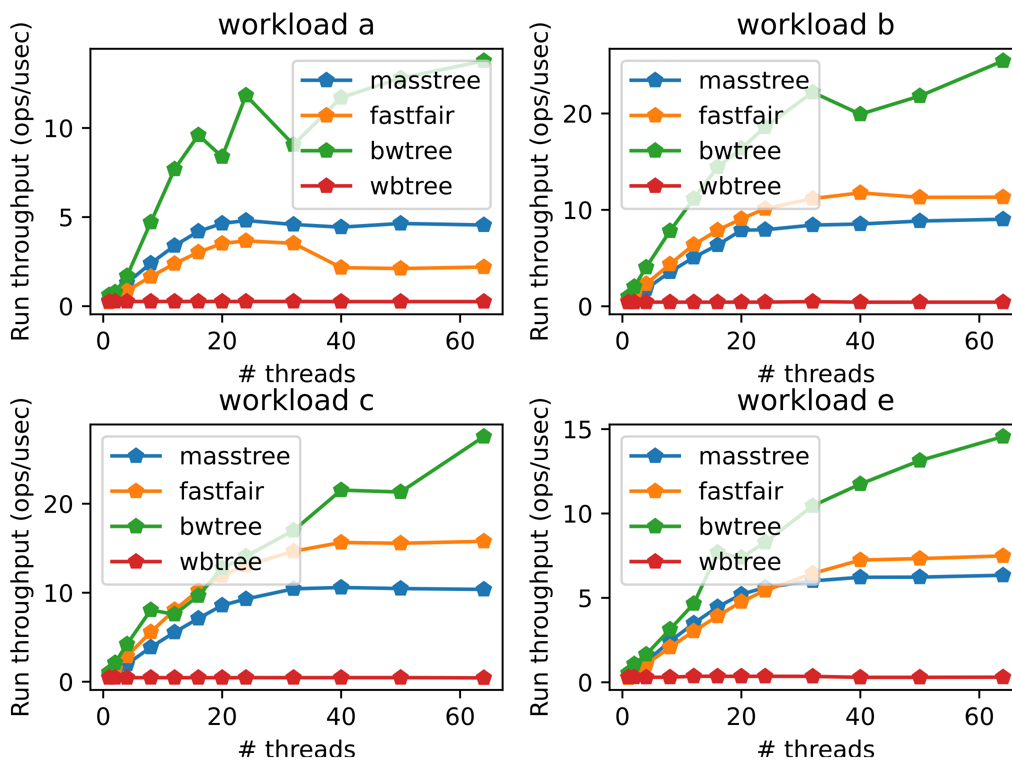
Πειραματική Αξιολόγηση με το YCSB

Το YCSB είναι ένα ευρέως χρησιμοποιούμενο benchmark στην αξιολόγηση key-value stores. Είναι παραμετροποιήσιμο ως προς την κατανομή των δεδομένων, το πλήθος νημάτων, το μείγμα των λειτουργιών(εισαγωγή, αναζήτηση, ενημέρωση, ερώτημα εύρους, διαγραφή) αλλά και ορίζει κάποια

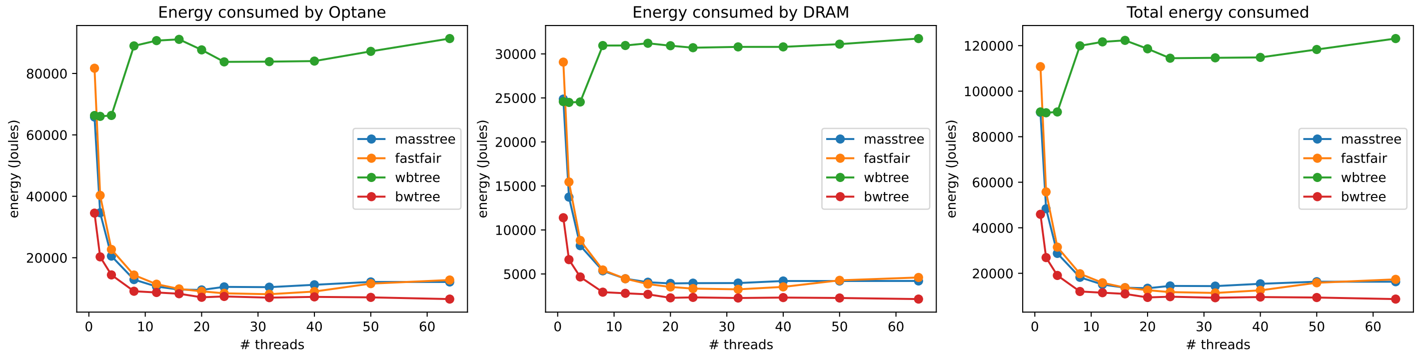
standard workloads, που μοντελοποιούν συγκεκριμένους τύπους cloud εφαρμογών. Από αυτά τα standard workloads έχουμε χρησιμοποιήσει τα A, B, C, E για ακέραια κλειδιά και κλειδιά-συμβολοσειρές που ακολουθούν ομοιόμορφη κατανομή.

Ακέραια κλειδιά των 8 bytes

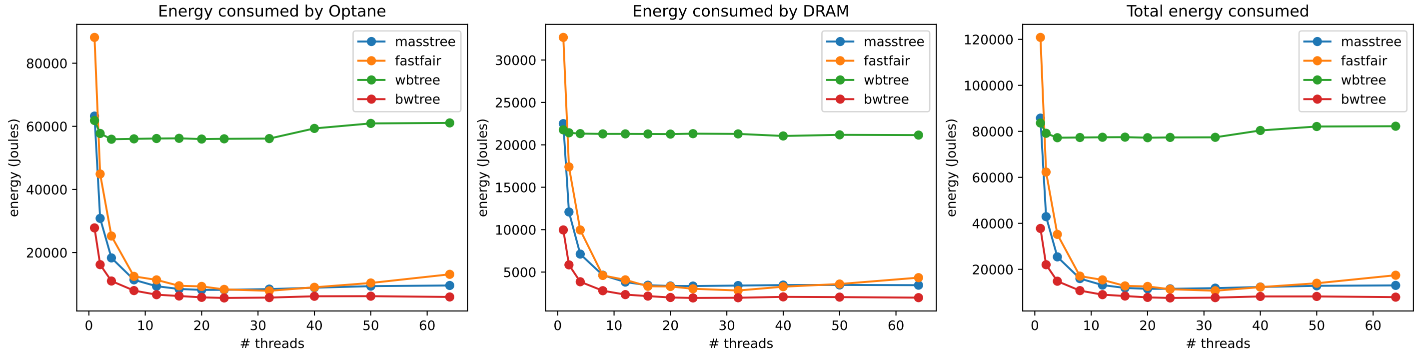
Για ακέραια κλειδιά, τα FastFair, Masstree εμφανίζουν ανταγωνιστικές επιδόσεις μεταξύ τους, αλλά το καλύτερο είναι το BwTree. Για 20 νήματα, που μέχρι αυτό το πλήθος νημάτων φαίνεται να κλιμακώνουν τα Masstree, FastFair, διαπιστώνουμε ότι το Masstree είναι 2.43 φορές πιο αργό από το BwTree και το FastFair 3.8 φορές πιο αργό για τη φάση load, για το workload A οι αντίστοιχες επιδόσεις είναι 1.8 και 2.4 φορές πιο αργά ενώ για το workload B, το FastFair είναι καλύτερο από το Masstree και είναι 1.8 φορές πιο αργό από το BwTree ενώ το Masstree είναι 2.07 φορές πιο αργό. Για την κατανάλωση ενέργειας παρατηρούμε και πάλι ότι όσο καλύτερη επίδοση έχει ένα index τόσο λιγότερη ενέργεια καταναλώνει, ωστόσο το Masstree αν και έχει χειρότερη επίδοση από το FastFair φαίνεται να καταναλώνει ελαφρώς λιγότερη ενέργεια. Οι προσβάσεις στη μνήμη δείχνουν να αυξάνονται καθώς αυξάνεται το πλήθος νημάτων. Επίσης το FastFair κάνει περισσότερες προσβάσεις στην NVM σε σύγκριση με το Masstree γεγονός που μπορεί να εξηγήσει την προηγούμενη παρατήρηση σε σχέση με την κατανάλωση ενέργειας.



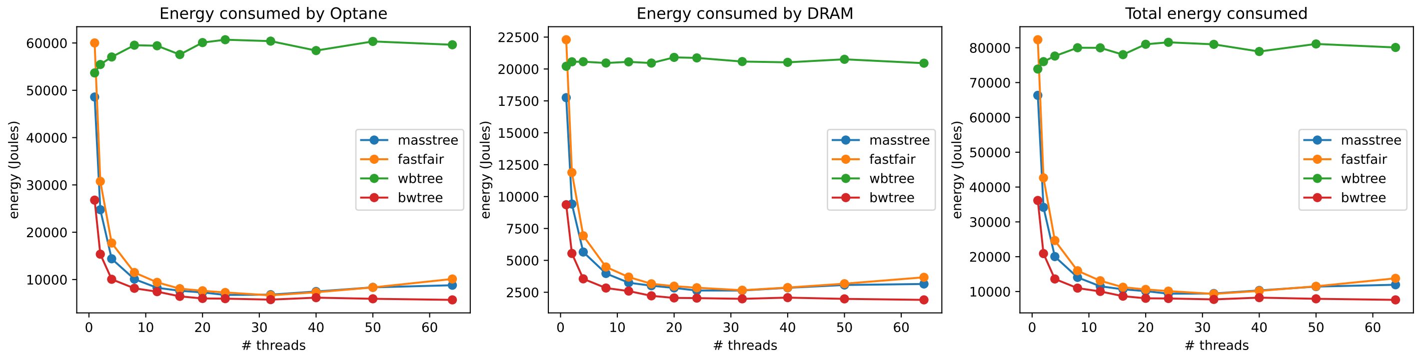
Εικόνα 6: YCSB run throughput for integer keys on Optane



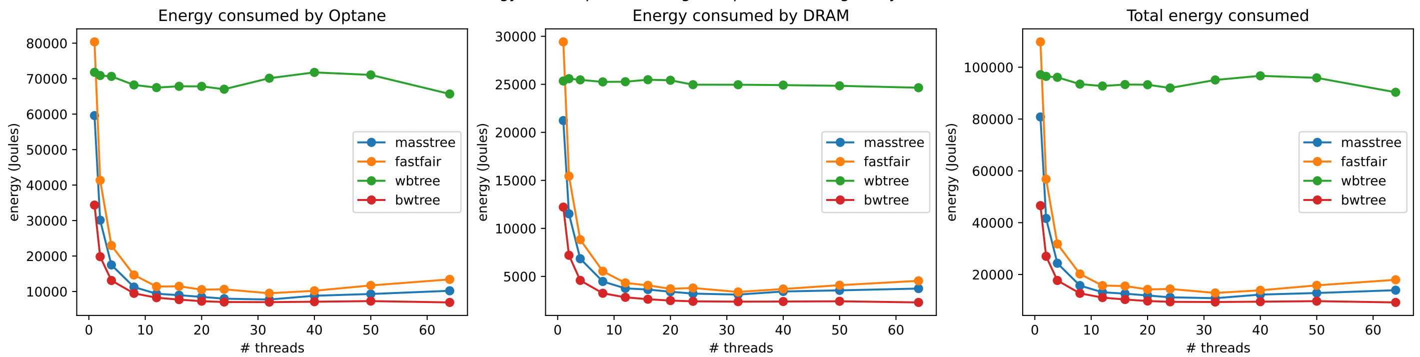
Εικόνα 7: YCSB energy consumption on Optane for workload A



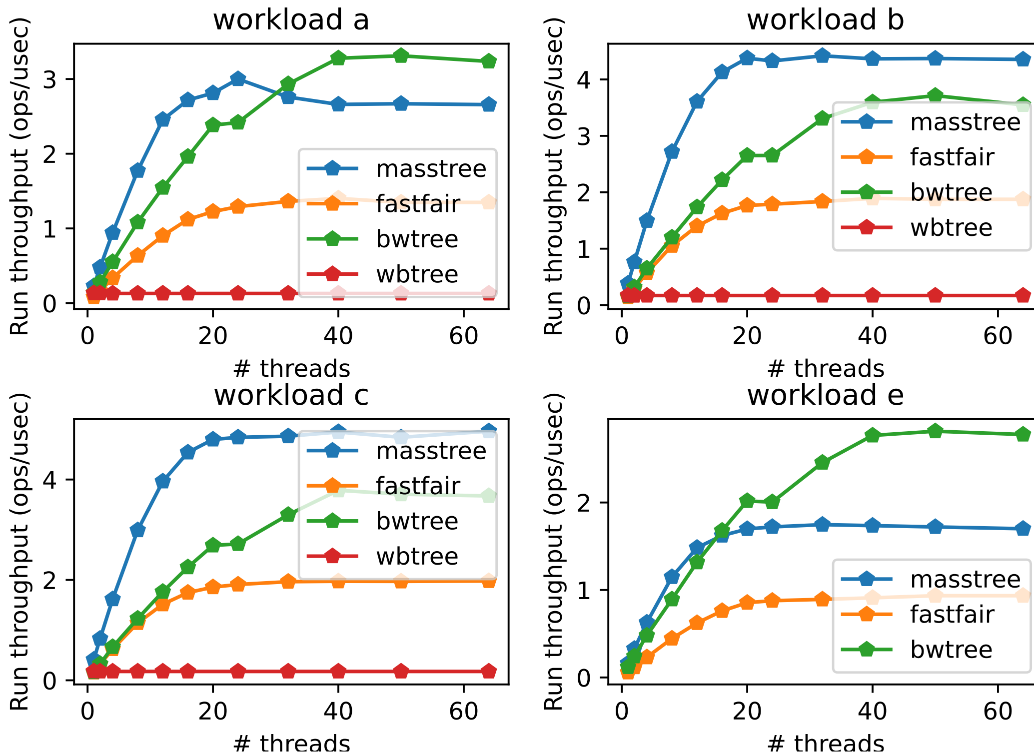
Εικόνα 8: YCSB energy consumption on Optane for workload B



Εικόνα 9: YCSB energy consumption on Optane for workload C



Εικόνα 10: YCSB energy consumption on Optane for workload E



Εικόνα 11: YCSB run throughput on Optane for string keys

Για κλειδιά-συμβολοσειρές την καλύτερη επίδοση έχει γενικώς το Masstree, του οποίου η αρχιτεκτονική είναι ένας συνδυασμός B+-tree και Trie, όπου το Trie είναι δομή βελτιστοποιημένη για τη διαχείριση συμβολοσειρών. Ανάλογα με το workload, το Masstree είναι από 1.18 έως 1.78 φορές καλύτερο από το BwTree, και από 2.3 έως 2.58 φορές καλύτερο από το FastFair. Διαφορές που εντοπίζουμε μεταξύ ακέραιων κλειδιών και κλειδιών συμβολοσειρών, είναι αρχικά ότι η επίδοση είναι χειρότερη για όλα τα indexes στην περίπτωση κλειδιών-συμβολοσειρών. Τη μικρότερη διαφορά έχει το Masstree, το οποίο όμως έχει επίδοση τουλάχιστον 1.65 φορές χειρότερη, ενώ τα BwTree, FastFair, WBtree έχουν επίδοση από 3.5 - 6.5, 2.9 - 6.4 και 2 - 2.5 φορές χειρότερη σε σχέση με ακέραια κλειδιά ανάλογα με το workload. Επίσης η κατανάλωση ενέργειας αυξάνεται, και είναι ενδεικτικά, για το workload A και 20 threads, 1.3 φορές μεγαλύτερη για το Masstree, 2.6 φορές μεγαλύτερη για το FastFair και 2 φορές μεγαλύτερη για το BwTree.

Σύνοψη και Προτεινόμενες Επεκτάσεις

Η συνεισφορά αυτής της εργασίας σε σχέση με προϋπάρχουσες αξιολογήσεις στη βιβλιογραφία, είναι πρώτον ότι χρησιμοποιεί ένα ακόμη benchmark για την αξιολόγηση, το TPC-C. Η υλοποίηση του benchmark είναι σε μορφή κατάλληλη ώστε να μπορεί να χρησιμοποιηθεί για την αξιολόγηση κι άλλων δομών εφόσον υλοποιούν ορισμένες συναρτήσεις σε μια κλάση wrapper. Δεύτερον, η εργασία μας εξετάζει επιπλέον την κατανάλωση ενέργειας, η οποία είναι μια μετρική που έχουν αγνοήσει προηγούμενες εργασίες.

Με το TPCC αξιολογήθηκε η επίδοση για transactional workloads. Διαπιστώσαμε ότι για το συγκεκριμένο benchmark είναι καλύτερη η επίδοση του Fast&Fair. Την καλύτερη επίδοση με

το TPCC σημείωσε το FastFair, έχοντας κατά 8.7%, 91% και 400% απόδοση από τα WBtree, P-Masstree, P-BwTree για 1 client, 1 warehouses, 750,000 transactions.

Επιπλέον με τις μετρήσεις μας με πολλά νήματα για το YCSB διαπιστώσαμε ότι τα περισσότερα indexes κάνουν scale μέχρι τα 20 threads, και ύστερα η απόδοση είτε παραμένει σταθερή είτε μειώνεται. Επίσης διαπιστώσαμε ότι κανένα index δεν αποδίδει εξίσου καλά για κλειδιά-συμβολοσειρές όσο για ακέραια κλειδιά.

Συνοπτικά, είδαμε ότι διαφορετικές δομές αποδίδουν καλύτερα με το TPCC και άλλες με τα workloads του YCSB. Όσον αφορά την κατανάλωση ενέργειας, παρατηρήσαμε ότι οι δομές που έχουν καλύτερη επίδοση έχουν και τη χαμηλότερη ενεργειακή κατανάλωση.

Υπάρχουν αρκετές προτάσεις για την επέκταση αυτής της εργασίας. Μία πρώτη επέκταση αφορά την υλοποίηση του TPCC, το οποίο θα μπορούσε να υλοποιεί ένα επίπεδο απομόνωσης λιγότερο αυστηρό από τη σειριοποιησιμότητα, καθώς όπως προέκυψε από τα πειραματικά αποτελέσματα, η απόδοση πέφτει αρκετά με το πλήθος των clients, και βασικός λόγος για αυτή την έντονη μείωση είναι ο ανταγωνισμός για τα κλειδώματα. Επίσης, η υλοποίησή μας για το TPCC είναι in-memory και το scaling factor(αριθμός warehouses) καθώς και το πλήθος συναλλαγών που εκτελούνται σχετικά μικρό. Θα είχε ενδιαφέρον να αξιοποιηθεί μια 'βιομηχανική' έκδοση του TPCC, όπως γίνεται για παράδειγμα στην εργασία των [9], οι οποίοι έχουν αξιοποιήσει το custom storage engine της MySQL και αξιολογήσει την επίδοση του συστήματός τους με το TPCC, χρησιμοποιώντας το ως MySQL plugin. Η αξιολόγησή μας ήταν επίσης περιορισμένη ως προς το πλήθος δομών που εξετάσαμε. Η συμπερίληψη περισσότερων indexes στην αξιολόγηση θα ήταν οπωσδήποτε μια χρήσιμη μελλοντική επέκταση. Επιπλέον, θα μπορούσαμε να εξερευνήσουμε την επίδραση του μηχανισμού συγχρονικότητας που χρησιμοποιεί κάθε δομή καθώς και να συμπεριλάβουμε περισσότερες παραμέτρους στις μετρήσεις, όπως cache misses, αριθμό cache line flush/fence operations για την εξαγωγή περισσότερων συμπερασμάτων για το πώς συμπεριφέρεται εσωτερικά κάθε δομή. Τέλος, η μεθοδολογία μας θα μπορούσε να αποτελέσει το πρώτο βήμα στην ανάπτυξη μιας μεθοδολογίας που να αυτοματοποιεί την επιλογή του κατάλληλου index structure για ένα ετερογενές σύστημα DRAM/NVM, ανάλογα με τον τύπο του workload και τις μετρικές προς βελτιστοιποίηση.

Chapter 1

Introduction

Persistent memory (PMEM) is a relatively new addition to the memory hierarchy, sitting between DRAM and flash-based storage and offering data persistence and direct (random) access (byte addressability) at close to DRAM speeds. It has greater capacity and lower cost per byte than DRAM and also lower energy consumption. This technology opens up new potential for building scalable, low-latency and high throughput, instantly recoverable applications with reduced total cost of ownership.

Database systems in particular, both memory-oriented and disk-oriented, are among the applications that stand to profit from leveraging this memory technology in their design, since they are required to durably store large volumes of data while also achieving high performance and quick recovery. There are several components of a database system where NVM can be integrated to improve aspects such as capacity, query performance and recovery time. For example, Arulraj [10] has extensively studied the integration of NVM into the logging and recovery, storage and buffer management, and indexing components of a DBMS. In recent years, NoSQL systems and key-value stores are gaining popularity in the database domain, alongside relational databases, due to their simplicity and flexibility compared to relational counterparts, as they do not have to adhere to rigorous data formats. At its core, a key-value store can be considered an index structure combined with a memory allocator. In relational databases, index structures are crucial for performance, hence index structures are a core component of modern database systems.

In a traditional database system, designed for a DRAM/SSD memory architecture, index structures are usually placed in DRAM as their purpose is fast data retrieval. However, as DRAM has limited capacity and is volatile, this limits the amount of data that a volatile index can store and can additionally increase downtime during system recovery, as the index needs to be rebuilt. Persistent memory enables the design of index structures that maintain high performance, while being able to store larger volumes of data and recover instantly. For these reasons, the design of persistent memory index structures are a prominent area of research in the broader field of redesigning database systems for new heterogeneous memory systems.

The first memory product of the NVM type recently became commercially available by Intel with the release of its Optane DCPM modules, which are based on the 3DX-Point technology. Before the release of actual persistent memory, researchers had used emulation and assumed that it would essentially perform as slower, persistent DRAM. The advent of actual hardware proved that this assumption was wrong. Compared to DRAM, PM has great read/write asymmetry and a smaller bandwidth that is easily saturated, especially in the case of writes. Since then,

several publications have explored how the design of such persistent memory structures needs to be redefined, based on existing technology and its actual characteristics. Researchers have proposed new structures specifically designed for Optane, as well as ways to adapt volatile structures for Persistent Memory to turn them into persistent ones. However, it is not so clear how the new proposals compare against each other and with past designs based on emulation.

The goal of this thesis is to propose a methodology for evaluating the performance and energy consumption of different index structures designed for heterogeneous DRAM/NVM systems. Our contributions compared to the existing literature are twofold; firstly, we additionally take into account the energy consumption as an evaluation metric and secondly, we evaluate index performance under transactional workloads using the TPCC benchmark in addition to YCSB. Both are aspects that the existing literature for the most part ignores.

This thesis is structured as follows: In Chapter 2 we provide an informational background on the Intel Optane Persistent Memory Technology and database indexing structures. In Chapter 3 we provide an overview of the existing literature and state-of-the-art persistent memory indexing, in Chapter 4 we discuss our evaluation methodology, in Chapter 5 we present and discuss the results of the evaluation. Chapter 6 concludes this thesis and proposes ideas for future research.

Chapter 2

Theoretical Background

Modern computer systems comprise several components. These components can be arranged in a hierarchy based on their characteristics such as capacity, access speed, and cost. This hierarchy, often referred to as the memory hierarchy pyramid, consists of multiple levels, with smaller, faster, and more expensive components closer to the CPU and the top of the pyramid, while larger but slower, and cheaper components are placed further away from the processor and towards the bottom of the pyramid.

Traditionally, these components are classified into two categories, memory and storage devices: Memory is fast, byte-addressable, which means it can be directly accessed via Load/Store instructions and volatile, which means it requires constant refreshes to retain data, and loses its contents in the event of a system crash or loss of power supply. Additionally, it is more expensive than storage. Storage is higher-capacity, lower cost, and persistent, which means that data written to it will be maintained even upon power failure. Storage is also much slower to access and, unlike memory, is block-addressable, so data from storage can only be accessed at block granularity and not directly with Loads/Stores. Persistent memory is a recently developed type of memory that combines properties of both types, breaking this strict classification. With the advent of persistent memory, the updated memory/storage hierarchy is illustrated in Figure 2.1

We will discuss Persistent Memory more in depth in the rest of this chapter.

2.1 Persistent Memory

Persistent memory (PMEM) or non-volatile memory (NVM) or Storage-Class Memory (SCM) is an emerging storage technology that combines the byte-addressability of DRAM with the persistence of storage, while maintaining close-to-DRAM speed, aiming to bridge the gap between DRAM and flash-based storage.

Several NVM types exist, such as Phase Change Memory (PCM), Spin-Transfer Torque RAM (STT-RAM), resistive RAM(ReRAM) and 3D X-Point, offering generally much higher memory density, much lower cost-per-bit and standby power consumption than DRAM [12]. Another notable property is the limited endurance compared to DRAM. Recently, Intel released its Optane DC Persistent Memory[13], which is the first commercially available persistent memory. In the remainder of this document, we often refer to it as simply Optane. A comprehensive

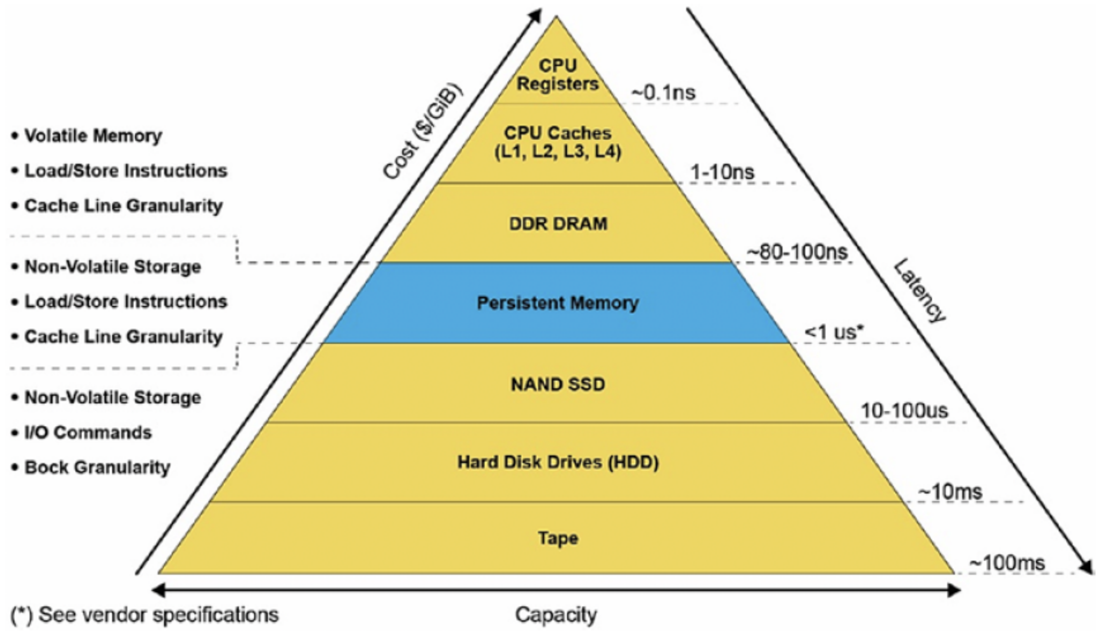


Figure 2.1: Storage/Memory hierarchy pyramid. Adapted from [11]

comparison of the different features of Non-Volatile Memory technologies is shown in Table 2.1

Memory Technology	Read Latency(ns)	Write Latency (ns)	Write Endurance	Standby Power
Flash SSD	25,000	200,000	10^5	zero
DRAM	80	80	$>10^{16}$	Fresh Power
PCM	50-80	150-1000	10^8	zero
STT-RAM	6	13	10^{15}	zero
ReRAM	10	50	10^{11}	zero
Intel Optane DCPMM	169 (Sequential), 305 (Random)	90	10^8	zero

Table 2.1: Different Features of NVMM Technologies. Adapted from [12]

2.2 Intel Optane DC PMEM

Optane PMEM is based on the 3D-XPoint technology. 3D-XPoint is based on modifying electrical resistance of a material using heat to change the state between crystalline and amorphous. Because each cell does not require a transistor, its density is around four times higher than DRAM [14].

Optane is connected to the processor in the same way that traditional DRAM DIMMs are: The Optane DIMM sits on the memory bus, and connects to the processor's integrated Memory Controller (iMC). The iMC is part of the CPU and its purpose is to manage data flow to and from the computer's main memory. DRAM and DCPM DIMMs are attached to the memory controller's memory channels. Only Intel's *Cascade Lake* CPUs and later can support the Optane DIMM. Each processor contains one or two processor dies and there are two iMCs per processor die, and each iMC supports three channels. Therefore, in total, a processor die

can support up to six Optane DIMMs across its two iMCs[15].

Persistence Domains A hardware system's *persistent domain* or *power-fail protected domain* is the set of areas for which it is safe to assume that, once data is located within them, will not be lost in the event of power failure[11]. For each Optane DIMM, the iMC maintains two buffer structures for reads and writes called the read pending queue(RPQ) and write pending queue(WPQ) respectively. The WPQ is especially important for persistence as it is part of the Asynchronous DRAM Refresh domain(ADR), together with persistent memory, and it is critical for ensuring the persistence of stores. Data to be written to a memory device is (after cache evictions) placed into the memory controller's write buffers where it is then written to the memory device [15, 11]. The ADR does not include the CPU caches.
The enhanced ADR persistence domain (eADR) also includes the CPU caches in the persistence domain, which enables persistence once data reaches the CPU cache. This means no flush instructions are needed for persistence. However the need for SFENCE instructions is not eliminated by eADR as it's still necessary to guarantee the correct ordering of writes, which SFENCE does by serializing store operations. It must be noted that eADR is not available prior to Intel's 200-series Optane DCPMM and Skylake platforms [2]

While the memory controller communicates with the NVM device with the DDR-T protocol using 64 byte cache line sized blocks, the internal page size of the NVM device, or more simply its access granularity is 256 bytes due to density requirements and physical space limits. Loads and stores (reads and writes) that are smaller than this 256 byte granularity waste bandwidth as they have the same latency as a 256 B access. Yang et al. [15] refer to this attribute as *XPLine* in their work. They also provide an overview of the Optane DIMM (shown in Figure 2.2):

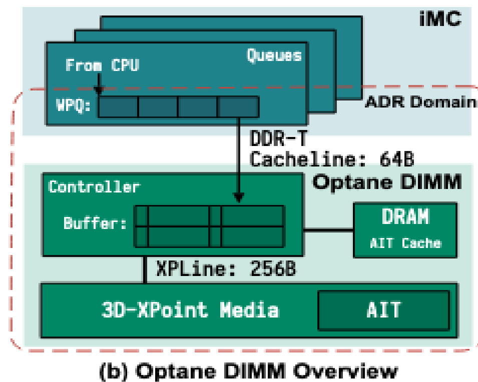


Figure 2.2: Optane DIMM Overview. Adapted from [15]

Each Optane DIMM also maintains buffers inside the DIMM-controller. There is a small write-combining buffer present on the on-DIMM controller that merges adjacent writes. [16] refer to this buffer as the read-modify-write(RMW) buffer, while [15] name it *XPBuffer*. Both specify its size as 16KB. This means that contiguous writes smaller than 256 B do not incur write amplification. However, non-contiguous stores smaller than 256 B do, as they trigger a read-modify-write operation in the controller, also reducing memory bandwidth [17].

Performance Characteristics Several studies have attempted to derive the performance characteristics of Optane. Important points in memory/storage performance characterization are latency, throughput and bandwidth.

Latency
The latency measurements of different studies are collected in Table 2.2.

Latency (ns)	Publication
305 (random read)	Izraelevitz[18]
160 (sequential read)	Izraelevitz[18]
450 (random read)	Benson[19]
50 (sequential read)	Benson[19]
403 (random read, app-direct)	[20]
382 (random read, memory mode)	[20]

Table 2.2: Reported Optane latency

Bandwidth
Izraelevitz et al[18] report that, "for a single Optane DC PMM, its max read bandwidth is *6.6 GB/s*, whereas its max write bandwidth is *2.3 GB/s*". As bandwidth increases with increasing number of DIMMs, the maximum bandwidth available is for the case of 6 DIMMs. Existing publications have reported the following bandwidth measurements (sequential read is the fastest among sequential read/write, and random read/write), listed in Table 2.3. There is

Bandwidth (GB/s)	# threads	Publication
40 (sequential read)	32	Benson[19]
39.4 (sequential read)	17	Izraelevitz[18]
13.9 (sequential write)	4	Izraelevitz[18]
13.9 (sequential write)	4	[20]

Table 2.3: Reported Optane bandwidth

some discrepancy in the reported number of threads to achieve the maximum bandwidth, but researchers report the same maximum values. There is also some discrepancy in the reported latency values, but other studies support the findings of Izraelevitz et al.

Operation Modes Optane has two modes of operation: the Memory Mode and the Application Direct Mode. In the Memory Mode mode, the DCPMM is used as a volatile extension of DRAM, acting as a large L4 cache without persistence support. This extends the capacity of DRAM, but also introduces additional access overhead for cached data, around 10%.[20] Persistence is enabled in the App-Direct mode. In the App-Direct mode, the DCPMM is exposed to the operating system as a separate persistent device. This can be mapped directly into the application's virtual memory space for example with mmap, or used with a file system. To guarantee persistence in the App-Direct mode, the application programmer must flush cache lines to PMEM by using e.g. the clwb instruction. As the compiler and OS can re-order instructions for better performance, it is necessary to explicitly avoid this behavior by issuing an sfence instruction which guarantees that the write to PMEM was completed and not reordered [19].

2.3 Programming Persistent Memory & PMDK

2.3.1 Operating system support for persistent memory

In order to provide support for integrating persistent memory in applications, operating systems like Windows and Linux have been extended with an NVDIMM driver, and a persistent memory aware filesystem. The NVIDIMM driver is required to make persistent memory detectable, and exposes it to the operating system and applications as a fast block storage device. A persistent memory aware filesystem is optimized for persistent memory since it can bypass the I/O subsystem and use persistent memory directly as byte addressable load/store memory (instead of using the block driver in the I/O subsystem, as it would for a regular block device, although that too is possible). This ability is referred to as Direct Access or DAX. DAX not only eliminates the I/O costs but also enables smaller read/write granularity, as it is no longer necessary to read and write entire blocks. [11] The two possible access methods are illustrated in Figure 2.3 below.

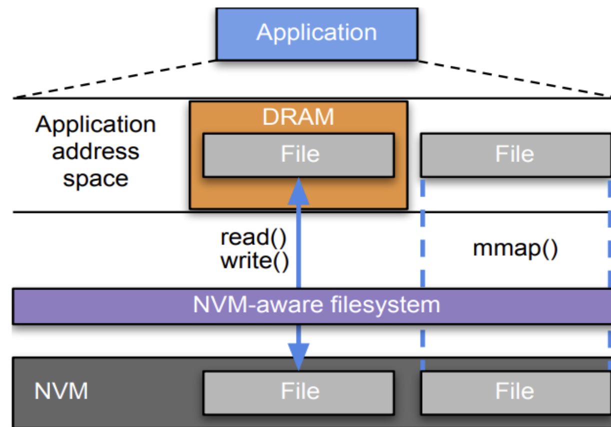


Figure 2.3: Basic Access Methods to NVM device. Adapted from [21]

Instruction Set Architecture (ISA) support As mentioned before, the path from CPU to NVM is long and for the most part volatile. In addition to this, modern CPUs support out-of-order execution. Therefore, certain instructions are necessary to ensure that data from the volatile CPU caches gets to NVM and in the correct order to guarantee consistency. In this section we mention the relevant instructions that are employed by data structure designed for persistent memory to guarantee the consistency of their data.

- **clflush** (Flush Cache Line) Evicts a cacheline and writes it back to memory. It is a synchronous instruction and does not require a memory fence to be serialized.

- **clflushopt (followed by sfence)** (Flush Cache Line Optimized) The asynchronous version of clflush. Not ordered with writes so it has improved throughput.

- **clwb (followed by sfence)** (Cache Line Write Back) This instruction writes back a cache-line to memory, but without evicting it from the CPU cache, which makes it preferable for performance reasons when it is to be accessed again soon after eviction.

- **mfence/sfence** Memory barriers. Sfence: memory barrier that serializes all pending stores. Performs a serializing operation on all store-to-memory instructions that were issued prior to this instruction. Mfence is similar to sfence, but it also serializes all pending loads as well as stores while sfence only serializes pending stores.

- **non-temporal stores (movnt, _mm512_stream_si512)** Non-temporal stores bypass the cache by writing to a special buffer, which is evicted either when it is full, or when an sfence is issued, therefore an sfence is still required to ensure the stores have reached the persistence domain.

An in-depth performance evaluation and comparison of cache line flush instruction variants and non-temporal stores has been conducted by van Renen et al in [20]. Both Scargall and van Renen advise developers to use primarily CLWB and non-temporal stores, falling back to other options if these are unavailable.

2.3.2 Challenges of Programming Persistent Memory

Compared to traditional programming, NVM introduces unique properties and with those also a new set of programming challenges. Some of the key challenges include:

Data Consistency As already mentioned, data that is directly accessed on NVM cache-resident before it reaches persistence. The cache however may at any time and in any order evict a cache line. Therefore it is necessary to ensure explicit ordering and flush of the data to the persistence domain to guarantee its consistency. This is achieved through the usage of cacheline flushing instructions in combination with memory fences, which prevent subsequent writes from completing before preceding writes.

Data Recovery When a program restarts, (after a crash for example), its virtual address space cannot be assumed to be the same as it was for the previous execution, due to the Address Space Layout Randomization (ASLR) feature used by most operating systems[11]. Since NVM is mapped into virtual address space and addressed using virtual pointers, like DRAM, it is necessary to have a reliable way of accessing and recovering data stored in NVM. Storing

Persistent Memory Leaks Memory leaks are an issue even with traditional programming with volatile memory: when a program fails to release dynamically allocated memory, this causes memory consumption to grow unnecessarily with execution. However, once execution finishes, these memory resources are freed. This is not the case with persistent memory; such memory leaks are persisted even after execution finishes, since the contents of NVM are retained. Persistent memory leaks can increase long-term memory consumption and can lead the In addition to this, with NVM there is a new class of persistent memory leaks as well. These leaks occur during persistent memory allocation: if the allocator marks the memory as allocated, before the data is written, a crash will result in the allocator seeing this allocation, but the application still requiring to write data to persistent memory, thus creating a memory leak.

Partial/Torn Writes Modern CPUs only support 8-byte atomic writes. This means that if there is a crash while an aligned 8-byte store is in progress, upon recovery these 8 bytes will contain either the old or the new contents. However, writes can be more than 8 bytes long. In that case, the application that needs to write more than 8-bytes must leverage the 8-byte failure atomicity supported by the hardware. A common solution to this problem is to use (up to 8 bytes) flags that can be written atomically, to indicate whether a larger write operation

has completed.

2.3.3 Persistent Memory Development Kit (PMDK)

The PMDK is a set of open-source libraries made available by Intel to facilitate software development for persistent memory. These libraries are meant to insulate application developers from the complexities of the hardware and to keep them from having to research and implement code specific to each platform or device. They are mentioned here briefly since the are building blocks for many of the applications encountered in the literature review and in particular, the index structures developed for persistent memory. It is worth mentioning especially:

- **libpmem**. Libpmem is a library based on the direct access feature (DAX), which provides a low-level platform-independent interface to PMEM.
- **libpmemobj**. Libpmemobj in particular is designed to address the challenges described in Section 2.3.2. It is a transactional object store, offering transaction support for persistent memory, handling persistent memory allocation and management, and internally it relies on the low-level PMEM support provided by libpmem.
- **libpmemobj-cpp**. This library provides C++ bidings for libpmemobj, allowing developers to use C++ for persistent memory programming to take advantage of C++ features such as the Standard Template Library (STL) and other useful C++-specific features and idioms. In fact all the indexes evaluated in this thesis are implemented in C++.

2.4 Linearizability

The concept of linearizability was introduced by Herhily[22] in 1989 and it is a correctness condition for data structures concurrently accessed by multiple processes. It enables us to reason about concurrent operations on a data structure as if they were performed sequentially, by a single process, with each operation taking effect instantly at a single point in time. The point in an algorithm where the operation is treated as "taking effect" by all other operations is known as a linearization point. Linearizability is the standard consistency model for DRAM concurrent data structures. Izraelevitz et al. in [23] have introduced the concept of *durable linearizability*, which extends the concept of linearizability to account for crashes, and which is the standard consistency model for PMEM indexes. Similar to how linearizability requires all operations to appear to execute atomically in some legal total order consistent with their real-time execution order, durable linearizability requires the same to also hold for the state recovered after a crash: it should correspond to some legal execution of a subsequence of the operations before the crash containing at least all the operations that completed before the crash.

2.5 Database Indexes

Indexing plays a significant role in improving the performance of the database. In the case of key-value stores, which are a popular form of database nowadays, as mentioned in the introduction, the index is one of the two basic components that make up the key-value store. The

purpose of the index is fast data retrieval. For that reason, in both traditional database architectures (main-memory oriented and storage oriented) the index is usually placed in DRAM, which is fast to access.

One of the key properties of a database system is durability, which is the guarantee that a committed transaction will remain committed even in the event of a system failure, such as a crash or power outage. The changes made by the transaction must survive and remain visible despite the system failure. To achieve that, some persistent storage medium is employed. NVM's byte addressability allows for low-latency loads and stores. At the same time, NVM is persistent, which means that unlike DRAM, all writes to NVM are potentially durable. Therefore, a database tuple or metadata stored on NVM can directly be accessed after a system crash, significantly speeding the recovery process by saving much rebuild/loading time. This makes bulding persistent memory indexes attractive, as there is no need for reconstruction after a crash. Additionally, with NVM having lareger capacity than DRAM, it is possible to build larger index structures to index more data at comparable speed.

There are several types of indexes used in relational databases and key-value stores, each taking into account different parameters to optimize for. In the remainder of this chapter we present the classic architecture of B+-trees, LSM-trees, skiplists, tries and hashtables, as these data structures are commonly used as building blocks in the literature we reviewed. In this thesis we focused on evaluating B+-tree based range indexes, which are among the most representative and commonly used in database systems.

2.5.1 B+-Tree

A B+-tree is a self-balancing m-ary tree. It consists of a root, inner, and leaf nodes. Unlike a B-tree, it only stores keys in the root and inner nodes, and values are stored only in the last layer, the leaf nodes. (This property is leveraged in the selective persistence implemented by many persistent indexes). A typical B+-tree node is shown in Figure 2.4. Leaf, inner and root nodes share the same structure, where P_i is a pointer and K_i is a key. It holds true that for i, j such that $i < j, K_i < k_j$. For $i = 1, 2, ...n - 1$ the pointer P_i points to a record, with a corresponding key value of K_i. The pointer P_n has a special function. It is a *link* pointer that is used to connect the records in the order defined by the search key. In the case of internal nodes it points to a record on the next level, whereas for the leaves it points to a sibling leaf.

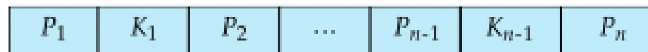


Figure 2.4: Typical B+-tree node

Figure 2.5 shows an example of a B+-tree.

Additional important parameters of B+-trees are the order of the tree d and its fill factor f. Each node has a number of entries that ranges between d and 2d, enforcing a minimum 50% occupancy at all times. Fill factor is the percentage of slots in the B+-tree that are occupied (filled with data). This is usually kept less than 100% to allow for quick inserts, otherwise, a

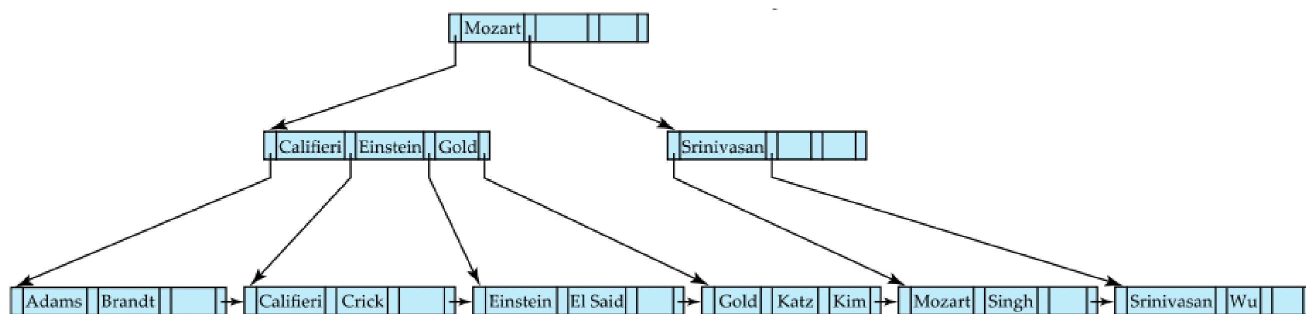


Figure 2.5: A B+-tree example

new node would have to be created much more often. Fanout is also defined as the inverse of the fill factor.

Point Lookup operations A lookup operation starts at the root of the tree. The target key is compared with the keys stored in the current node, and based on the comparison, the appropriate child node to traverse to is determined: If the target key is less than the smallest key in the node, then follow the leftmost child pointer to the children. If the target key is greater than the largest key, follow the rightmost pointer. Otherwise, select the pointer P_i such that, $K_{i-1} \leq K < K_i$ and follow it to the next level of the tree. This process is repeated at each inner node of the tree until reaching the appropriate leaf node. There, the leaf node is searched until the target key is either located in the leaf and the corresponding value is returned, or the target key is not found and the fact the target was not found is indicated. One observation is that without getting to the leaf node, it is not possible to tell whether a key exists in the B+-tree.

Insert operations An insert operation first locates the leaf where the new record must be inserted, the same way a lookup operation does, then inserts it in that leaf, in the appropriate place to maintain the ordering of keys, shifting other keys if necessary. An insert may cause the leaf to have more keys than allowed, therefore a split operation is performed to enforce that structural properties of the B+-tree are not violated. The split may be propagated up the tree to the root level, causing the tree to grow in height by one level.

Delete operations

Delete operations also begin by locating the leaf containing the record to be deleted, and then delete the entry from the leaf node. If after deletion, the node contains fewer entries than acceptable, it is first attempted to "borrow" records from a sibling leaf. If that is not possible, then the leaf is merged with a sibling. Deletions are also propagated up the tree, as described later in the section *Node Merge and Node Split Operations*

Range Lookup operations A range lookup operation retrieves the values corresponding to a given range of keys. It is almost identical to a point lookup operation using the lower bound of the range. When reaching the appropriate leaf node, the range lookup operation then performs a sequential iteration through the leaf node keys, starting at the key corresponding to the lower bound of the search. While the current key is not greater than the range end, the range lookup retrieves the value corresponding to the current key, and moves to the next key to the right. It's possible to cross to the right sibling leaf through the sibling pointer.

Node Merge and Node Split Operations These are the Structural Modification Operations (SMOs) taking place in B+-trees. Node Split operations are triggered by Insert operations that cause the leaf node they are inserting into to exceed its capacity. Then the leaf is split in two,

by creating a new sibling leaf, keeping the first $\lceil n/2 \rceil$ entries in the old node and moving the following entries to the new node. The parent node must also be updated to contain the new key and to point to the new child. Inserting the new key into the parent node can cause additional split of the parent node, which might need to be propagated up the tree reaching the root. A split of the root adds an extra level to the tree.
Similarly, a Delete operation can trigger a Node Merge, if borrowing from siblings is not possible. In that case, entries from the right leaf are moved to the left, the empty right child is deleted, and the parent node must be updated: the entry pointing to the recently deleted child must be removed. This removal can cause the parent node to contain less than the minimum number of keys, so it may have to be merged with a sibling node as well. This can recursively lead to node merges up until the root level, and it is possible to reduce the height of the tree by one level.

Concurrency in B+-trees

It is very important for an index to support concurrent operations in order to be scalable with increasing number of threads and remain performant. In B+-trees, the structural properties must be maintained at all times. Insert and delete operations though can cause structural modifications across several nodes of the tree, as mentioned. The simplest concurrency protocol would be to lock the entire tree. However, that is not performant. Therefore, fine-grained solutions for concurrency are desirable, yet not trivial to implement.

B-link Trees A basic "building-block" of B+-tree concurrency are *B-link trees*[24]. They are a prevalent solution for concurrency in B+-trees that provides lock-free searches and guarantees that at most three nodes will be locked at any given time by other operations that can cause structural modifications. Briefly, B-link trees allow for delayed updates to the parent, also relaxing the requirement to maintain structural properties of the tree at all times a little bit, as will be described below. B-link trees are B+-trees with a few additional properties: They maintain pointers from the left sibling to its right sibling, at all levels of the tree (whereas for B+-trees, this is only true for leaf nodes). The last key in an internal node is known as the *high key*. Searches to the tree take no locks.

B-link tree search Searches do not take any locks. If a split happens concurrently with a search, and the high key of the current node is smaller than the search key, this could mean that a split has occurred and the parent node had not been updated yet, so following the link pointer, the right sibling is searched. If the key is found in the sibling, the search succeeds, otherwise fails.

B-link tree insertion The leaf to insert into is located through the search process described above. Once found, it is locked. In the worst case, the node will need to be split. A new sibling node is created, and the link pointer of the original leaf node is updated to point to the new node. Then, the parent node is locked, as it will need to be updated with a key and pointer to the new child node, and locks on the child nodes are released. At most three nodes are locked.
For brevity, we skip the discussion of delete and range lookup operations. An additional thing to note about B-link trees is that they delay updates to parent nodes: If a leaf node overflows (or underflows) as a result of an Insert (or Delete), then a new sibling node is added (removed) and the high-key pointer accordingly updated. The parent is updated at a later time, when a write operation occurs on it.

2.5.2 LSM-Tree

The B+-tree structure is not ideal for applications that need to support a very large number of random writes/inserts per second[25]. A popular alternative to B+-tree is the LSM-Tree, which is widely used in modern key-value stores and NoSQL database systems. The Log-Structured Merge-Tree (LSM-Tree) was originally proposed in a 1996 paper[26], as a disk-based index structure that can support low-cost real-time indexing for insert/delete heavy workloads. In its simplest form, illustrated in Figure 2.6, it comprises two components, one small memory-resident component and one large, disk-based component, but can also have a multi-component architecture. Each component is larger than the previous one, usually with exponentially increasing capacity between consecutive layers, resembling a tiered tree-like structure.

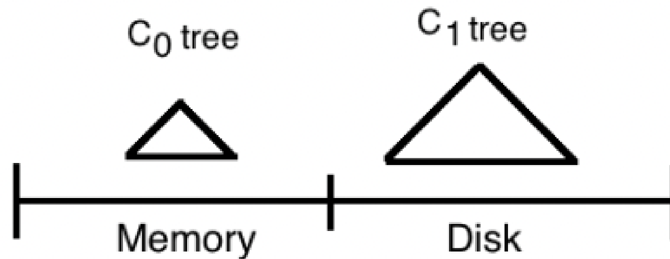


Figure 2.6: Two-component LSM-Tree architecture

The memory-resident component (C_0) is also often referred to as *MemTable* (Memory Table) in literature[17, 27]. For the remainder of this document, we also refer to the in-memory component of an LSM-Tree as MemTable.

LSM-Trees are optimized for write performance: As data is inserted into the tree, it is first placed in the MemTable (and also written to a write-ahead-log to ensure durability). This way, write operations are fast since they involve the memory and do not incur I/O costs using the disk, which has much higher latency. When the MemTable (or in general, a component) size reaches capacity, the data it contains is sorted, batched together and then flushed to the next level, in a *rolling merge* process.

Unlike a B+-Tree, which is described by a specific implementation, an LSM tree can be considered more a conceptual than a structural directive. Each layer doesn't have a specific format and its components may be implemented in a variety of ways, using different data structures[28]. The original paper[26] implemented its components as B-trees. An LSM-tree index supports the following operations:

Insert (write) operations Inserts into LSM trees simply add the new record into the MemTable as mentioned.

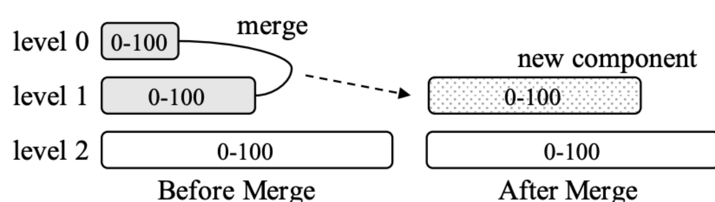
Point Lookup operations LSM-trees are out-of-place update structures, and therefore, a key-value pair may be found across several components. Lookup operations in LSM-trees usually have to search across multiple components for a specific key. A point lookup (searching for a single key-value pair) must retrieve the latest value corresponding to the given key. To do that, the operation simply searches for the given key across all components of the LSM-tree, starting at the newest component and moving down the tree to the oldest, stopping once the first match is found. This is possible since newest data is contained in topmost components.

Delete operations Deletes in an LSM-Tree are in fact treated as inserts, by inserting a special *"tombstone"* value for the respective key-to-be-deleted, which indicates deletion. On a lookup, if the most recent (top level) entry for a key is a tombstone, then that means the key is not present in the index.

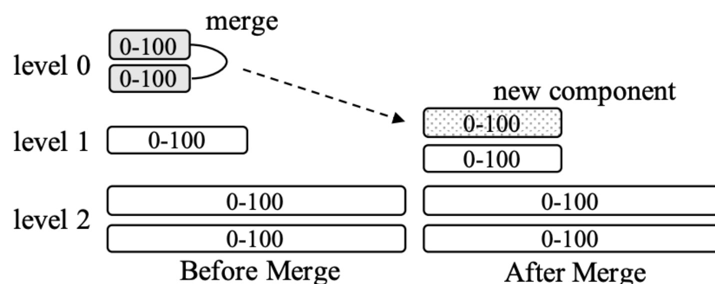
Range Lookup operations A range lookup in LSM-trees has to perform *reconciliation*, that is, to find the latest version of each key. In order to do that, the operation will search multiple components simultaneously, using a priority queue such as min-heap to keep the latest version of each key-value pair available.

Merge(compaction) operations As components accumulate over time, read performance tends to degrade since the number of components it has to search increases. To address that, disk components are periodically compacted(merged). Two main compaction schemes exist, although in practice combinations of both are used. The first is leveling, the other is size-tiering, illustrated in Figure 2.7. In a leveling merge (this technique is employed for example by RocksDB[29] and LevelDB), each level has a fixed size. Key-value pairs from adjacent levels are merge-sorted, then inserted into the lower level. This causes write amplification, as the lower level can be several times larger than the higher one [17] Leveling merge is more read optimized, as it reduces the number of components that must be searched during a read. In size-tiering, each level consists of multiple sublevels whose key ranges overlap. Key merge-sort operation is conducted among the sub-levels and the result key-value items are written as a new sub-level of its lower level compaction is better for writes since it reduces merge frequency.

Concurrency in LSM-trees For concurrency control, LSM-trees need to handle concurrent reads and writes, and also deal with concurrent flush and merge operations. The main challenges with concurrency in LSM-trees are concurrent merge and flush operations. During a flush operation, the MemTable to-be-flushed becomes read-only, in order to still support reads, and new writes will go into a new MemTable. As mentioned, each component of an LSM-tree does not have a specific implementation, although it is common for components to be implemented as B+-trees, so it is up to the respective implementation to correctly handle concurrency internally.



(a) Leveling Merge Policy: one component per level



(b) Tiering Merge Policy: up to T components per level

Figure 2.7: LSM-Tree merge policies. Adapted from [28]

2.5.3 Skiplist

A skiplist is a probabilistic in-memory data structure first proposed by Pugh[30], as an alternative to B-trees, which performs similarly to them but without the need for explicit rebalancing, as inserts and updates do not require rotation or relocation but use probabilistic rebalancing instead. A skiplist is shown in Figure 2.8. It is essentially a linked list of nodes of different heights. The height of each node is determined by a random function and computed during Skiplists have inferior worst-case performance compared to B+-trees, but their average time complexity is the same due to their probabilistic nature. Their space complexity is also better than that of B+-trees. They are relatively simple to implement and support all operations supported by B+-trees: inserts, deletes, point as well as range lookups.

Point Lookups A search operation in a skiplist begins from the highest level of the header, following the highest level pointer of the current node, until it reaches a node where the key is larger than the search key. Then it returns to the predecessor and follows the next-level pointer. This process is repeated until the search key is located.

Inserts An insert operation follows the same process as a point lookup, to locate the insertion point. Then a new node is created, the height h of which is determined randomly. All the predecessor nodes of height up to h are linked to the new node, and the successor nodes are linked as well.

Deletes Deletes follow the same process as lookups and inserts to locate the deleted key, then once the corresponding node is located, forward pointers of the removed node are placed to predecessor nodes on each level.

Range Lookups A range lookup operation starts as a point lookup operation for the lower key in the range. Once it reaches the corresponding node, it follows the bottom level pointers of each node that is within the range, retrieving the corresponding values until there is no longer a node within the range.

Concurrent skiplists Concurrent skiplists can be implemented by using an additional *fully_linked* flag [31], that determines whether or not the node pointers are fully updated. This flag can be set using compare-and-swap. Atomically setting the flag is necessary because the pointers have to be updated on multiple levels to fully restore the skiplist structure.

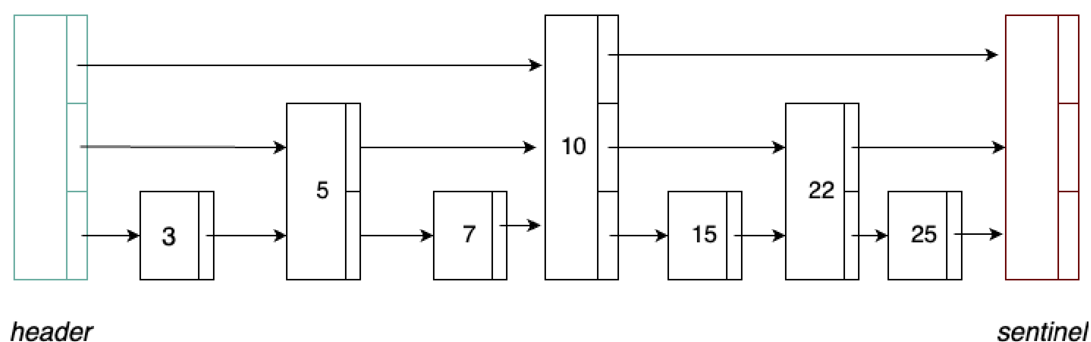


Figure 2.8: Skiplist

2.5.4 Trie & Radix Tree

Trie, also known as Prefix Tree or Digital Tree, is an m-ary tree that is commonly used to store and retrieve strings efficiently. The word "trie" comes from the word retrieval, which accurately

describes its purpose. A trie is shown in Figure 2.9 Its shape only depends on the key space and key lenghts, and not on existing keys or their insertion order, which means that rebalancing is not a requirement for Tries, and all operations have O(l) complexity, where l is the key length. A Radix Tree is a special type of Trie. It is also known as a "Compact Prefix Trie" because its main difference from a Trie is that instead of a single character, each node stores a string prefix that may consist of multiple characters. For this reason, a Radix tree can require rebalancing with inserts and deletes.

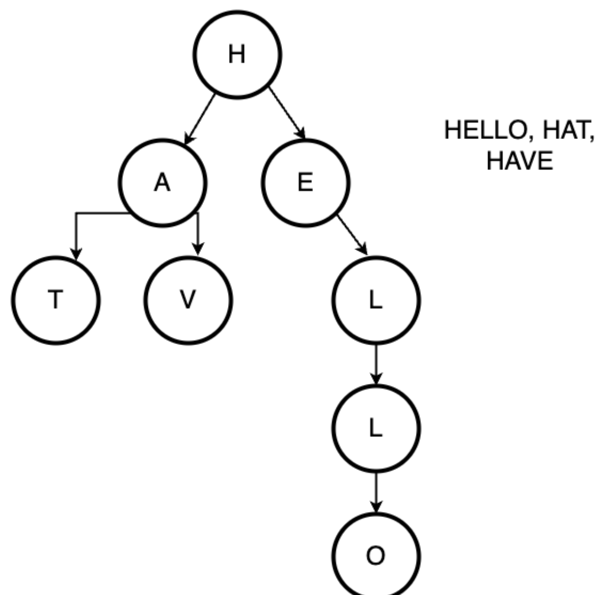


Figure 2.9: Trie

2.5.5 Hash index

A hash index is a data structure that uses a hash function to map keys / values to a fixed-size array called a hash table, and it performs very well for point queries. We have not included them in our evaluation, but listed here for completeness, as there are works adapting them for persistent memory, as well as evaluation works assessing the efficiency of proposed solutions[3]

Chapter 3

Literature Review

Non-Volatile Memory integration into Database Systems is an area of active research, before and after the advent of actual hardware; as mentioned in the Introduction, several components of a database can benefit from the use of NVM. For example, Shanbhag et al.[32] study how well in-memory databases perform when large datasets are stored in Persistent Memory instead of DRAM; van Renen et al.[33] propose a 3-tier (DRAM, NVM, SSD) architecture (in accordance with the guidelines of [34]), where buffer management is adapted to keep warm pages in NVM, allowing the Database System to maintain its performance with increasing dataset sizes; Chen et al.[35] also explore the integration of Persistent Memory in an in-memory database engine in the context of real-time feature extraction applications and On-Line Decision Augmentation: by replacing a volatile skiplist in the critical path with a persistent one, they remove the need to sync logs from the critical path of the application, reducing response times, minimizing recovery time and reducing total cost of ownership.

In this thesis, we have focused on the indexing components of a database system and how they can leverage Persistent Memory. While NVM has features that make it attractive for use in database indexing, it is not trivial to design efficient index structures for database applications that take into account and leverage the unique properties of NVM. Simply moving data structures from DRAM or SSD to NVM is not beneficial, as it is a different class in the storage hierarchy with its own performance characteristics. Several designs have been proposed by researchers, aiming to leverage the unique properties of NVM. Some studies have also proposed general guidelines for persistent index design; however, there is no one-fits-all-usecases solution.

In this chapter we give an overview of recent work on persistent index design, presenting the main architectural components and implementation choices of the proposed indexes. We also reference other evaluation works for in-memory and persistent index structures. The architecture of the indexes we included in our evaluation is discussed in further detail in chapter (insert section number when appropriate). as well as other index structures that were not evaluated in this thesis.

3.1 Challenges of persistent index structure design

Researchers have identified the following challenges in designing index structures for persistent memory[1, 36, 37]:

C1: persist primitive (clwb, clflush, sfence) is necessary to ensure persistence and

consistency of writes, but also expensive (high latency, device wear) As CPU caches are volatile, one has to ensure that data is flushed to PM by using instructions like clflush or clwb. Furthermore, writes must be performed in a certain order for consistency, which is ensured by fence instructions like sfence. Fencing also drains the CPU store buffer and stalls the execution pipeline.
C2: Structural Modification Overhead (SMO) By this term is desginated the write amplification caused by the need to maintain metadata and structural properties of the index when inserting a new key-value pair. Any further write besides the inserted key-value pair incurs write amplification.
C3: PM Bandwidth scarcity The bandwidth of persistent memory is limited and more easily saturated compared to DRAM. Especially for machines with a smaller number of NVDIMMs, maximum PM bandwidth is even smaller. Therefore, persistent data structures must be designed to not exhaust the available bandwidth.

3.2 Recent Index Designs for Optane

Persistent index designs proposed for PM can be roughly classified into three categories according to their architecture: B+-tree based, Trie based, and hybrid [36, 2]

3.2.1 BzTree

BzTree[38] is a PMEM-only, latch free B+-tree structure. It uses PMwCAS (persistent multi-word compare and swap) for concurrency. Both inner and leaf nodes of BzTree are stored in PMEM. BzTree applies copy-on-write to keep inner nodes immutable except for updates to child pointers. Inserting to a parent node causes it to be replaced with a new one that contains the new key. Then, an update in the grandparent node is conducted to point to the new parent node. Splits can propagate up to the root and grow the tree. Records in inner nodes are always sorted, while records in leaf nodes are not. Initially,records are inserted to the free space serially. Periodically leaf nodes get consolidated (sorted) and subsequent inserts may continue to insert into the free space serially. After searching the sorted area(using binary search), the tree must linearly search the unsorted area to get correct result. The design rationale is that inner nodes are not updated as often as leaf nodes and should be search-optimized; leaf nodes, however, need to be write-optimized.

3.2.2 Masstree

Masstree[39] is a "highly-concurrent, cache-efficient trie-like concatenation of B+-tree nodes". It was designed to provide high performance even for variable-length keys, potentially with long common key prefixes, which is enabled by its trie-like design. Essentially Masstree is a trie, in which each node of the trie is a B+-tree, indexed by a different 8-byte slice of a key. The concurrency scheme used is write exclusion with lock-free readers, with readers retrying if they observe an inconsistent state, indicated by a version number. To reduce write operations, inserts into leaf nodes in Masstree which do not cause an SMO to the tree, do not reorder the nodes. Instead, a new key-value pair is simply appended to the node, in an unordered fashion, and a separate 8-byte permutation table (maintained per-node) is atomically updated. The internal nodes in Masstree, however, do maintain the sorted order, hence operations leading to

SMOs employ a non-atomic, key-shifting algorithm to reorder records. Masstree was converted by the authors of RECIPE to a persistent version.

3.2.3 PACTree

PACTree[37] is a PM-only, hybrid index combining trie and B+-trees. It consists of a search layer and a data layer, plus a structural modification operation log. The search layer is implemented as a trie, based on ART and using Read-optimized write exclusion (ROWEX) for concurrency, while the data layer is implemented as a doubly-linked list of unsorted B+-tree-style leaf nodes. Each node in the list contains an *anchor key*, which is the minimum key in the node, as well as a fingerprint array and a permutation array to speedup point queries and scans. In order to prevent them from becoming a scalability bottleneck, SMOs in PACTree only update the data layer; the SMO is logged in the corresponding log and the update of the search layer is performed asynchronously by a dedicated background thread that replays the SMO log. Additionally, PACTree was implemented to take into account NUMA effects, by using separate pools for the data and search layers and logs in each NUMA node.

3.2.4 Fast&Fair

Fast&Fair[40] is a concurrent B+-tree that provides lock-free reads. It is composed of two algorithms, Fast (Failure Atomic ShifT) and Fair (Failure Atomic In-place Rebalance). Fast is used to insert the keys within a node of the B+-tree by performing atomic shift operations to maintain the sorted order of the keys. Fast also reduces the number of clflush instructions, because flushes are called only when crossing a cache line, and not for every array element shift. Because a shift operation requires updating both the key and its corresponding pointer, assuming 8-byte keys, a key could be atomically persisted and a crash could occur before its respective pointer is updated. This would leave the tree in an *endurable inconsistent state*, where a duplicate key appears in a node; however, because the B+-tree property of unique keys is violated, readers encountering this know to disregard a key that appears between two identical pointers. Fair avoids the use of logging (that is usually necessitated by SMOs in B+-trees), by exposing intermediate states to readers, who can detect and tolerate such inconsistencies without being blocked by writers. For concurrency, Fast&Fair provides lock-free reads, but uses mutexes for write operations.

3.2.5 P-BwTree

BwTree[41] is the volatile version of the P-Bwtree used in our experiments. BwTree is a highly concurrent B+-tree developed for new hardware platforms which is completely lock free, in order to scale efficiently on multi-core systems. It has a complex architecture based on B+-trees. Tree nodes are logical memory pages, each logical page having its own PID, which replaces the pointer in the traditional B+-tree. A logical page consists of a linked list, where each node of the list covers a certain key range and is indexed by its pid. A mapping table is used to keep track of the correspondence between logical pages and physical memory addresses of the head node of the composing linked lists. A logical page is updated using a "delta strategy" which prepends a new delta record to the original head node of the linked list of the logical page; then the mapping table is udpated with the new mapping using a single CAS instruction. This complex architecture is captured in Figure 3.1

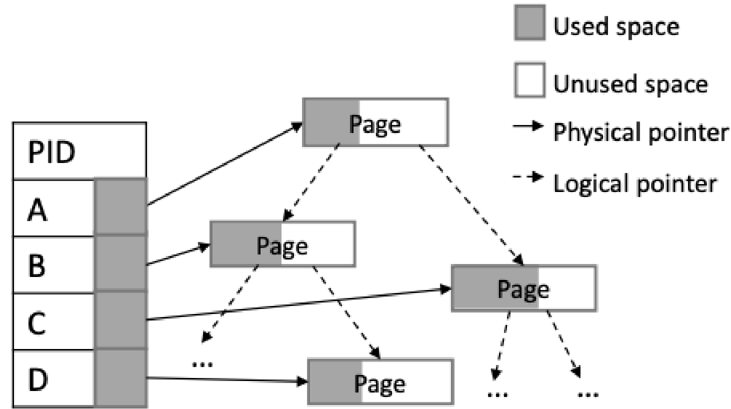


Figure 3.1: Architecture of BwTree (Adapted from [42])

BwTree was converted by the authors of RECIPE to a persistent version.

3.2.6 wBTree

wBTree[43] stands for write atomic B+-tree which optimizes the performance of insert and delete operations while striving to maintain good search performance. It aims to minimize the overhead introduced by expensive PMEM write operations and by cacheline flushes and achieves this by employing several methods: Keeping nodes unsorted. A new record is inserted into a free slot without any sorting occurring in-place, thus reducing PM writes. Two additional structures can be part of each node, a bitmap and an *indirection slot array*. The purpose of the bitmap is to indicate which entries are valid, and it is up to 8 bytes, therefore atomically updated. Hence, inserts and deletes that do not incur SMOs, need only update the bitmap, again reducing the need for additional writes. When operations do cause SMOs, wBTree uses undo/redo logging. However, the bitmap only guarantees atomic updates, and not sorted order, which is why in addition, the indirection slot array is used, and is updated instead of the leaf nodes to maintain ordering and enable faster lookups. In small nodes, the bitmap can be omitted and the indirection slot array serves to guarantee atomicity, indicate validity and maintain sorted order. The authors of wBTree mention that it can be extended to support variable length keys, through the use of pointers. It does that by storing 8-byte keys in the tree, which are actually pointers to the actual variable sized keys. In addition, wBtree is not a concurrent data structure.

3.2.7 DPTree

DPTree[36] employs an architecture reminiscent of a two-level LSM tree. Its volatile/DRAM resident memtable (L0) is called buffer tree and is a B+tree, while its persistent L1 called the base tree, employs selective persistence, comprising a DRAM component (radix tree) and a PM component (linked list of leaf nodes). Since the buffer-tree is DRAM resident, durability is ensured by the existence of a write-optimized redo log in PM. The decision to employ two levels is made to reduce read latency. The authors refer to it as the *dual-stage index architecture*. For concurrency, DPTree employs Optimistic Lock Coupling[44] to make the DRAM-resident buffer tree concurrent and enables concurrent logging by hash partitioning the log and implementing

each partition as a linked list of log pages. Tree components are synchronized by tracking active readers/writers to make sure they are accessing the correct version of the buffer tree.

3.2.8 ChameleonDB

Zhang et al have developed ChameleonDB[17], a key-value store for Optane PMEM. ChameleonDB's index architecture is based on LSM-trees. However, while an LSM-tree is an attractive option for deployment on PMEM due to its being optimized for writes, they notice that a multi-level LSM-tree is incapable of leveraging Optane's low read latency, and therefore propose an architecture that combines the strengths of LSM in terms of write performance with hashtables to speed up reads. The authors implement a sharded index, where each shard covers a range of the key-space. Its in-memory components are the MemTable of the LSM structure, plus the *Auxiliary Bypass Index (ABI)*. The ABI is an in-DRAM hash table that indexes all the keys in the next levels, except the last. This reduces the the read latency, as a read operation has to search at most 3 levels, only one of which is PMEM-resident: the MemTable, the ABI, and the last level of the LSM structure. Each LSM level consists of multiple fixed-size hash-tables. They employ Lazy Leveling compaction scheme, that is, size-tiered compactions for the intermediate LSM-levels and leveling compaction for the last level, thus balancing write amplification and read latency.

3.2.9 ViPer

ViPer, (standing for Volatile Index Persistent data) is an embedded DRAM-PMEM key-value store designed by Benson et al.[19]. The authors use a custom persistent memory allocator, to first memory map PMEM into virtual address space and then allocate memory in blocks. They maintain an in-DRAM hash index, the *Offset Map*, based on CCEH, which stores the page id and offset of a persistent memory block. They also employ fingerprinting, by keeping a hash of each key in the Offset Map.
In summary, the specific design choices of each reviewed persistent index are collected in Table 3.1

Index	Architecture	Node Structure	Concurrency	String keys
wBTree	B+-tree;PMEM-only	Unsorted, indirection slot array	Single-threaded	Pointer to key
FPTree	B+-tree; selective persistence; inner nodes in DRAM, leaf nodes in PMEM	Unsorted leaf nodes; fingerprints	HTM & locking	Pointer
BzTree	B+-tree;PMEM-only	Partially unsorted leaf; sorted inner nodes	lock-free(PMwCAS)	Inline
DPTree	selective persistence; B+-tree and inner trie in DRAM; trie leaf in PMEM	unsorted leaf; fingerprints; indirection; extra metadata	optimistic locking; async updates	Pointer
PACTree	Trie;PMEM-only (optional selective persistence)	Unsorted leaf; fingerprints; indirection	optimistic locking; async updates	Inline
FastFair	B+-tree;PMEM-only	Sorted nodes	Lock-free reads; blocking writes	Pointer
Masstree	Hybrid: trie-like concatenation of B+-trees;converted	unsorted leaf; sorted internal;	lock-free reads;write exclusion	Inline
BwTree	B+-tree;converted	logical pages; mapping table;deltas prepended to node;	non-blocking reads and writes	Inline

Table 3.1: Collective Table of design choices per index

3.3 Other evaluation works

Xie et al [42] have evaluated in-memory indexes Masstree, BwTree, Fast, ART, and PSL in a DRAM/SSD memory system, studying their performance in terms of query throughput and latency, scalability with increasing number of threads, memory consumption and cache miss rate, for diverse workloads. They evaluate skip-list based indexes (PSL), B+-tree based indexes (Masstree, BwTree), and trie-based indexes (ART, Masstree).

Regarding the performance of Masstree and BwTree, they find that they perform worse than the other structures due to their using linked lists instead of arrays, as arrays allow for cache alignment and SIMD processing, while linked lists allow for flexibility. They also scale less well for write workloads, compared to the other evaluated index structures. Also, BwTree, followed by Masstree, consume the most space; especially BwTree due to its delta-append logic for updates. For BwTree they observe that the write throughput drops considerably for write-only workloads, which they attribute to "contention caused by frequent fails of the hot blocks". Masstree is found to exhibit the least performance fluctuation for different kinds of workloads.

A limitation of their evaluation is that they only use the YCSB benchmark with integer keys with a fixed length of 4 bytes. That does not showcase the strengths of Masstree and BwTree, both structures aiming for flexibility by supporting keys of arbitrary length, unlike the other indexes. Another finding of their evaluation is that Hardware Transactional Memory (HTM) might not scale so well compared to other concurrency solutions if there are frequent concurrent exclusive accesses to a critical section, so it requires careful engineering.

Other evaluation works specific to Persistent Memory indexes are those of Lersch et al[1], He et al[2] and Hu et al[3].

As part of their evaluation work, Lersch et al [1] have developed PiBench[45], (Persistent Index Benchmarking framework), which is an effort to provide a unified framework for "highly customizable benchmarks, ruling out the impact of different benchmark implementations". They chose to evaluate persistent memory range indexes, that is indexes that support accessing values within a range, and in particular B+-tree index structures, as B+-trees are the most widely used index type in OLTP systems, they have been extensively studied and mature techniques have been developed for B+-trees in the context of persistent memory. The indexes used in the evaluation are NVTree, BzTree, FPTree (fingerprinting tree) and wBTree. They are chosen as representative of different concurrency schemes, node architecture and placement choices.
NVTree is lock-based, BzTree is lock-free and uses PMWCAS, FPTree employs HTM for its inner nodes and fine-grained locking for its leaf nodes. WBtree and BzTree place nodes in PM only, NVTree is PM-only but could also be hybrid (DRAM/PM), FPTree places inner nodes in DRAM and leaf nodes in PM. The metrics tracked in this work are throughput (operations/second), accesses to persistent memory, operation latency, cache misses, and number of cacheline flushes to persistent memory per tree operation (insert/update/delete). Their key findings are summarized below:

- **throughput** Placement of inner nodes in DRAM makes traversal faster and leads to higher performance for FPTree and NVTree. The best performer is FPTree, due to its optimizations: leveraging DRAM and fingerprinting.
- **per-operation performance** Lookup performance also affects other operations as first

they do a lookup - traversal. Insert performance is directly affected by number of flushes per insert, maintenance work per insert, and overhead of node splits. Update performance again depends on the number of writes and flushes to PM. Scan: reading less from PM does not compensate the overhead of sorting and filtering.

- HTM and PMwCAS (both optimistic concurrency schemes) are not scalable, as they are vulnerable to high contention. The finding regarding HTM is inline with the observations of [42] .
- **recovery time** As expected, PM only structures are able to recover instantly after a crash. Structures that employ selective persistence are much slower to recover, but in the case of clean shutdowns, inner nodes could be spilled to PM to enable fast restart. The wBTree recovery time is two orders of magnitude less than that of FPTree (the faster of DRAM-PM structs). Also, for DRAM-PM the recovery time grows linearly with the dataset size, while it is constant for PM only structures as all they have to do is retrieve the root object persistent pointer.

He et al. have relied on the previous work of [1] and extended it in [2], to include the impact of variable-length keys, PM allocator and NUMA effect in the evaluation. Indeed the impact of variable-length keys has not been studied by other evaluations. They evaluate DPTree, uTree, LB+Tree, ROART and PACTree (which are either newer, or focused on optimizing other metrics than throughput - uTree for latency) against FPTree which was found to be the best performing index in the pre-optane era. They find that proposals from the "pre-Optane era" employed optimizations that are beneficial even for Optane (such as fingerprints and selective persistence), while newly proposed indexes do not necessarily outperform older proposals. There are still aspects that are not adequately addressed by existing proposals, such as supporting variable-length keys, mitigating NUMA effects and efficiently managing PMEM. An interesting observation is that when running in DRAM without extra memory fences and cache line flushes, an index designed for PM may outperform well-tuned volatile indexes. This, combined with the evaluation findings from RECIPE and TIPS, which prove that DRAM indexes converted for PM can outperform PM-crafted indexes, suggests that there is potential for unifying index design for both types of memory.

Hu et al. perform similar evaluations on Persistent Memory hash indexes (CCEH, Level-hashing, Dash, PCLHT, SOFT, Clevel), leveraging and extending the Pibench benchmarking framework published by Lersch et al. Some of their key findings have been that:

- Random small writes, which are an inherent pattern in the design of hash tables, are the primary factor restricting performance of hash indexes on PMEM.
- Contrary to the observations made for range indexes, the impact of the persist primitive (cost of clflush/sfence) is in comparison very small, due to the smaller number of these instructions required in hash table operations.
- Fingerprinting is an optimization that works well for hash indexes.

3.4 Converting DRAM indexes to persistent memory indexes

Developing an index tailored for persistent memory is a challenging task, as mentioned in Section 3.1, and addressing issues like concurrency, persistent memory allocation without leaks

is not trivial. On the other hand, DRAM indexes have been studied in-depth over the course of many years, and are much more mature and optimized. As noted by [4, 7], it is much easier and less error-prone to convert a volatile index to its persistent counterpart, than it is to build a persistent index from scratch. For structures that meet certain criteria, it saves a lot of effort to try and take advantage of existing DRAM indexes, that are known to do their job very well, by converting them to persistent ones. This enables applications to avail of a large number of well-engineered indexes, but it also paves the way for porting applications from DRAM to NVM.

RECIPE[4] is a "principled approach for converting DRAM index structures to persistent memory index structures". The guidelines provided by RECIPE are not at the source level, but in practice indexes can be converted by only adding a few lines of code, when they meet one of the following conditions. The key observation the authors make is that volatile consistency of reads and writes has a persistent counterpart (consistency of reads and writes after a crash)

1. Reads are non-blocking, writes can be either blocking or non-blocking. Write operations are made visible to other threads using a hardware-atomic store. The solution is to add clflush and mfence/sfence instructions after each store (and each load for non-blocking writes because in their case, the order of the load/stores can not be guaranteed by a lock).

2. Reads and writes both non-blocking. Writers fix inconsistencies. Solution: add clflush and mfence/sfence after each load and store. Example: BwTree.

3. Non-blocking reads, and blocking writes.

After ensuring the persistence of stores by adding clflush and sfence instructions, the volatile allocations must also be converted to persistent memory allocations. The conversion method proposed by RECIPE is essentially to convert volatile memory allocations to persistent memory allocations, by using PMDK's libvmmalloc. Libvmmalloc transparently converts traditional dynamic allocation interfaces to work on a volatile memory pool built on a memory-mapped file on PMEM. Other frameworks have also been proposed to convert volatile index structures into persistent ones, such as PRONTO[6], NVTraverse[5] and more recently, TIPS[7]. Unlike RECIPE, it does not impose restrictions on the concurrency model, (RECIPE requires the concurrency model to be either lock-free, or fine-grained,) supporting any concurrency model.

PRONTO Pronto is a library that adds persistence to volatile data structures using *asynchronous semantic logging (ASL)*. Pronto requires that the converted volatile structure meets two conditions: that its modification methods do not read or write global variables, and that the structure is linearizable, if it is concurrent. Application developers can add persistence to their volatile structures by wrapping the data structure in a PersistentObject class and implementing appropriate wrapper methods for the structure's API and using the Pronto memory allocator instead of malloc/realloc/free. Pronto consists of three components: The ASL, which logs the persistent operations, a *volatile online image* of the data structure in DRAM, and a *persistent snapshot* of the data structure. Essentially PRONTO places the index on DRAM and logs the necessary operation on PMEM.

NVTraverse NVTraverse is a general transformation that takes a lock-free data structure that belongs to the *traversal data structure* class and transforms it into a PMEM implementation that is durably linearizable and highly efficient. It is applicable to lock-free data structures.

Chapter 4

Proposed Framework

In this chapter we describe our evaluation methodology and the evaluation framework we propose. We apply a systematic and extendable methodology to enable the effective evaluation of state-of-the-art indexes, which consists of four steps, as illustrated in Figure 4.1.

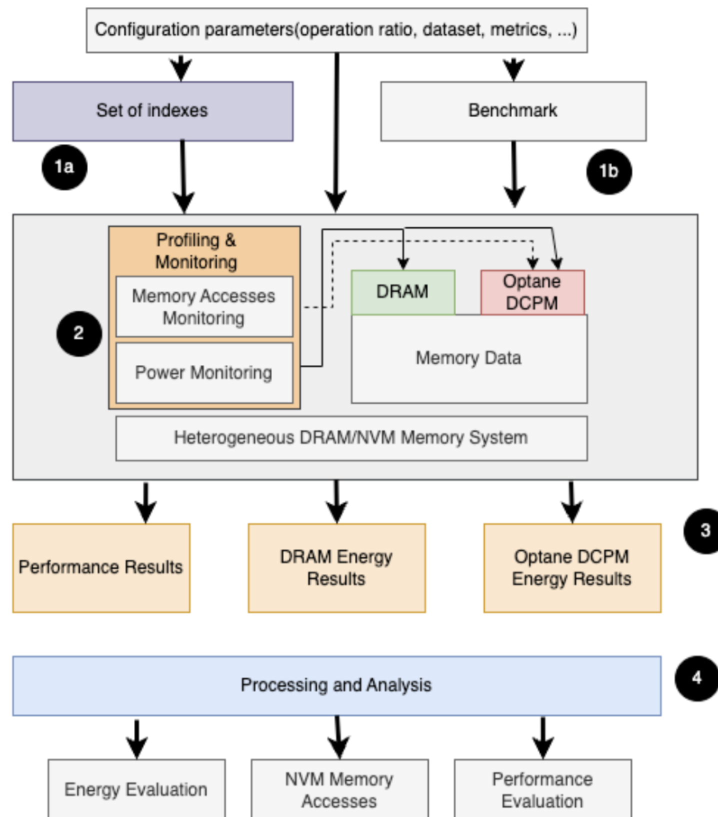


Figure 4.1: Evaluation Methodology Overview

On the one end of our methodology, we provide as input a set of indexes (Step 1a), a benchmark(Step 1b) which is configured by a set of parameters (for example, ratio of operations, data distribution) and a set of metrics of interest to the evaluation such as throughput and energy consumption. The benchmarks we used were TPC-C[46] and YCSB[47]. The set of indexes we evaluated was based on the open-source implementations provided by [4] and consists of the following implementations: P-BwTree, P-Masstree, WBtree and Fast&Fair. Each index's architecture is described in detail in the respective subsection of Chapter 3.

The Profiling & Monitoring Component of the methodology (Step 2) is responsible for

power and performance monitoring and it is based on Intel's Processor Counter Monitor (PCM), a tool that provides run-time low-level system metrics and allows to sample energy/power over the main memory DIMMs through hardware sensors [48]. The raw measurements are then grouped and processed (❸ and ❹ respectively) in order to provide the throughput, total memory accesses and energy consumption results for each evaluation experiment. Experiments are setup with shell scripts and results processing is done with Python scripts.

Energy consumption is monitored using Intel's *pcm-power* tool[49], which reports measurements for both DRAM and Optane. We have used a sampling rate of 10 milliseconds in all experiments. Accesses to the Optane Persistent Memory medium are measured by the *ipmctl* utility.

For the TPC-C Benchmark, we have provided our own in-memory implementation, which we describe in the following sections.

4.1 TPC-C Benchmark Specification

TPC-C is an industry standard benchmark for evaluating the performance of OLTP (On-Line Transaction Processing) systems and their ability to handle transactional workloads. In summary, it represents a wholesale business scenario, simulating a set of transactions that are commonly encountered in real-world business operations, such as order placement, inventory management, and financial accounting. It measures the performance of the DBMS by executing a mix of transactions involving multiple users, each performing tasks like order processing, payment processing, and database updates. The TPC-C schema consists of nine tables and five procedures that simulate a warehouse-centric processing application and its E-R diagram is shown in Figure 4.2

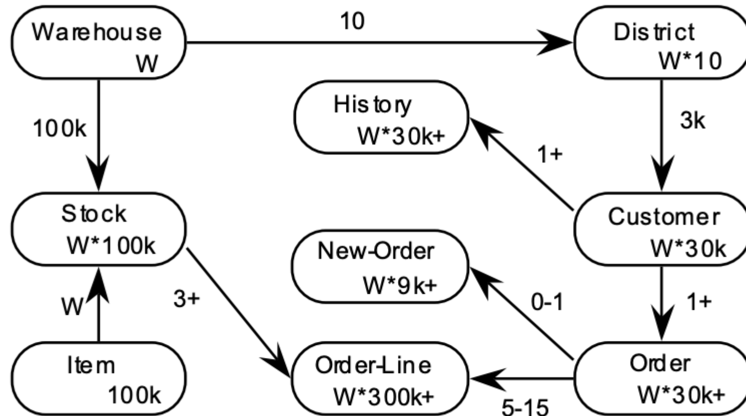


Figure 4.2: TPCC Entity-Relationship Diagram. Adapted from [46]

The workload mix defined by TPCC consists of the following five distinct transaction types:

- New-order (45%) Represents the submission of a new customer order.
- Payment (43%) Simulates a payment transaction, where a customer pays for an existing order
- Delivery (4%) Updates the status of an order to indicate delivery.

- Order-status (4%) Retrieves the status of a customer's latest order. Order-status is a read-only transaction.
- Stock-level (4%) Calculates the current stock level for a specific item. This is also a read-only transaction.

As shown in Figure 4.2, the Scale Factor (SF) of the TPC-C benchmark is the number of warehouses. The TPC-C benchmark consists mostly of transactions that insert or access new records (i.e. NewOrder) and older records are almost never accessed. There is therefore a strong temporal skew built into the semantics of the benchmark. Only a subset of tables are increasing in size, and the rest are static. [10].

4.2 In-memory TPC-C Benchmark implementation

For the TPC-C performance experiments, we based our implementation of the benchmark on an existing C++ implementation[8], to which we made the necessary extensions to support multiple clients and to enable evaluation of any index as a plug-in. The benchmark we used is an in-memory implementation of the schema and workload specified in [46]. For randomness, we treat the transaction mix as a deck of cards, with multiple clients(terminals) picking transactions(cards) from the deck. We randomly shuffle the deck in the beginning of execution. Like most implementations in research, we do not implement think times. A class diagram for the implementation is shown in Figure 4.3. This is an overview of the implementation; the diagram is not exhaustive as several helper methods are omitted here.

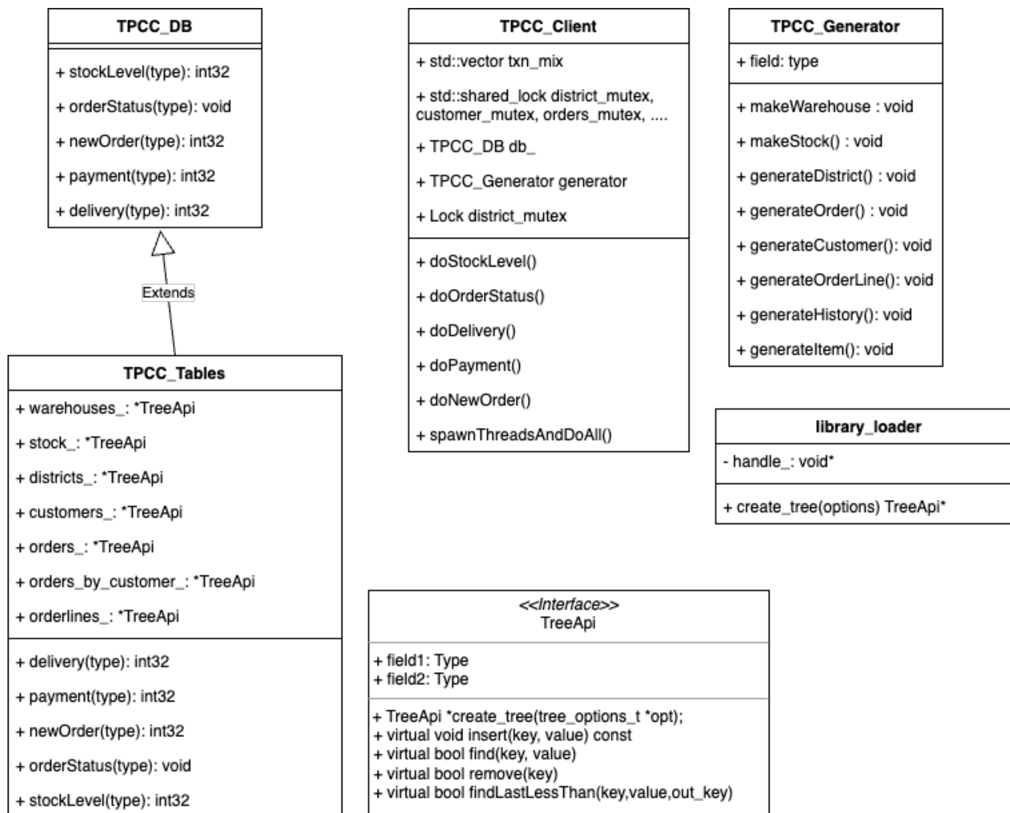


Figure 4.3: TPCC UML Diagram

TPCCDB This class defines the structs representing each TPCC relation, such as Order, OrderLine etc. It acts as an interface for transaction implementation, by defining virtual functions

for each transaction type; these are actually implemented in the TPCCTables class.

TPCCClient This class is the one responsible for issuing transactions. The logic for transaction isolation is also implemented in this class; among its members are locks for each TPC-C table, that transactions must request at their start. It also has a vector member corresponding to the transaction mix; for multiple clients, each client is a different thread that is allocated a distinct range of the transaction mix to execute.

TPCCTables The methods corresponding to transactions are implemented here. Also, here we define the underlying tables for the TPCC relations: warehouses_, stock_,districts_, customers_, orders_, orders_by_customer_, orderlines_ : these are all instances of the TreeApi class, which is a wrapper around the evaluated index.

TPCCGenerator This class implements the methods responsible for populating the TPCC database, pre-filling the tables before transaction execution begins.

Our benchmark defines an abstract class, TreeApi, that defines the following operations: insert, find, remove, and a custom operation, findLastLessThan, as shown in Listing 4.1. The custom operation is similar to the Scan operation of the YCSB benchmark, as its functionality is similar: It starts at a given key in the index and scans the records in order, retrieving the value that corresponds to the last key that is smaller than a given value. It is required in order to support the OrderStatus transaction.

```
1  class TreeApi;
2  extern "C" TreeApi *create_tree(tree_options_t *opt);
3
4  class TreeApi
5  {
6  public:
7      virtual ~TreeApi(){};
8      virtual void insert(const void *Nkey, const void *Nvalue) {};
9      virtual bool find(const void *key, void *value = nullptr) const { return 0; };
10     virtual bool remove(const void *key) {return 0;};
11     virtual bool findLastLessThan(const void *key, void *value = nullptr, void *
       out_key = nullptr) const {return 0;};
12 };
```

Listing 4.1: TPC-C API

Each evaluated index must implement these functions, extending the TreeApi class in a wrapper class. Then at runtime, the index being benchmarked with TPC-C will be loaded as a shared library.

4.2.1 Multiple clients support

TPC-C is a transactional benchmark, therefore in order to support multiple clients / multiple concurrent sessions, we must also provide transaction isolation. We have chosen to implement the strongest isolation level, serializability, with strict two-phase locking (2PL). An advantage of strict 2PL is deadlock prevention, which made it an attractive option for our in-memory implementation of the benchmark. One disadvantage though is the increased resource contention and reduced concurrency. Our implementation of serializability is based on the `std::shared_mutex` class of C++. Each table (districts, orderlines, stock, history, customerByName, ordersByCustomer, newOrder, Orders, Customer, Warehouse) has its own shared mutex. Depending on the type of operation each transaction performs with a table, it either takes a read or write lock on

it. In order to make sure that we do not run into a deadlock, we ensured that locks are taken and released in a specific order in each transaction.

4.2.2 Mapping of TPCC operations to tree operations

The TPC-C benchmark defines transactions, whereas the YCSB benchmark defines certain ratios of tree operations such as inserts, lookups, scans. We have quantified how the TPCC implementation we used translates to basic tree operations, by using a counter for each tree for each type of operation and summing the results after the run finishes. For 10 warehouses, 1 million transactions and a single client, a total of 10,249,106 inserts were performed, 37,200,424 lookups and 40,000 range scans.

4.3 Evaluated Indexes

The indexes included in the evaluation were **P-BwTree, P-Masstree, WBtree, Fast & Fair**. Their architecture is described in the respective subsections of Chapter 3. We obtained the open-source implementations of all index structures from [4]. In most cases, we had to make some modifications to the source code to support the findLastLessThan operation. This operation is similar to a range scan, where instead of determining the number of keys, starting at a certain key, we determine the end key for the scan. As the range scan supported by most indexes is of the form RangeScan(startkey, number), we had to modify the source code of most indexes to implement our version of a RangeScan. We implemented a custom rangeScan operation for all indexes except P-BwTree based on their existing implementations of scan and rangeScan. P-BwTree's implementation provided iterators, therefore making it easy to implement our own custom scan without modifying the source code.

Additionally we considered for the evaluation DPTree, PACTree, FPTree and BzTree, but we leave the integration of these indexes with TPC-C as future work due to the modifications/bugfixes to their source code required for a correct wrapper class implementation. (We found bugs in the implementation of DPTree and PACTree; PACTree was experiencing frequent crashes whereas DPTree searches returned incorrect values. We did not find bugs with FPTree and BzTree but they required more extensive source code modification).

Chapter 5

Evaluation Results

This chapter presents the main contribution of this thesis, which are our evaluation results.

5.1 Server characteristics and configuration

Experiments were conducted on a machine with the following characteristics and configuration:

Sockets	2
threads per core	2
cores per socket	20
microarch	Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz
L1/L2/L3 cache	1.3MBL1i & 1.3MBL1d / 40MB / 55 MB
Total DRAM	128GB
Total NVMM	1536 GB (6x256 GB)
GNU/Linux Distro & Kernel	Ubuntu 20.04.2 LTS (Focal Fossa) 5.4.0-121-generic

Table 5.1: Evaluation platform specifications

5.2 Evaluation metrics

- **TPCC transaction throughput** reported in transactions per second.
- **YCSB operation throughput** reported in operations per microsecond.
- **Total energy consumption** Energy consumption is a metric that existing literature has not widely explored, focusing mostly on operation throughput instead. Katsaragakis et al. have provided some insights into the energy consumption of hybrid DRAM-NVM applications that use Intel Optane in memory mode[50, 51], App-Direct mode [52], as well as the energy consumption of database index structures[48]. We rely on the methodology used in [48] to measure energy consumption.
- **Optane read/write accesses** As mentioned, Optane memory accesses are measured by the *ipmctl* utility, by monitoring the read and write accesses to each Optane DIMM.

5.3 TPC-C

TPC-C consists of two phases, a load phase in which data is loaded into the database and tables are populated with "old data"; we report the total number of keys initially loaded in Table 5.2 (we show the average over 6 runs). The load phase of the TPCC implementation we used is single threaded. From the table we can see the load time is the time required to load about half a million keys per warehouse, with a single thread.

# Warehouses	Total Keys Loaded
1	489424
2	979997
4	1957960
8	3917608
10	4900255

Table 5.2: TPCC number of keys loaded

5.3.1 Load phase

5.3.1.1 Load time

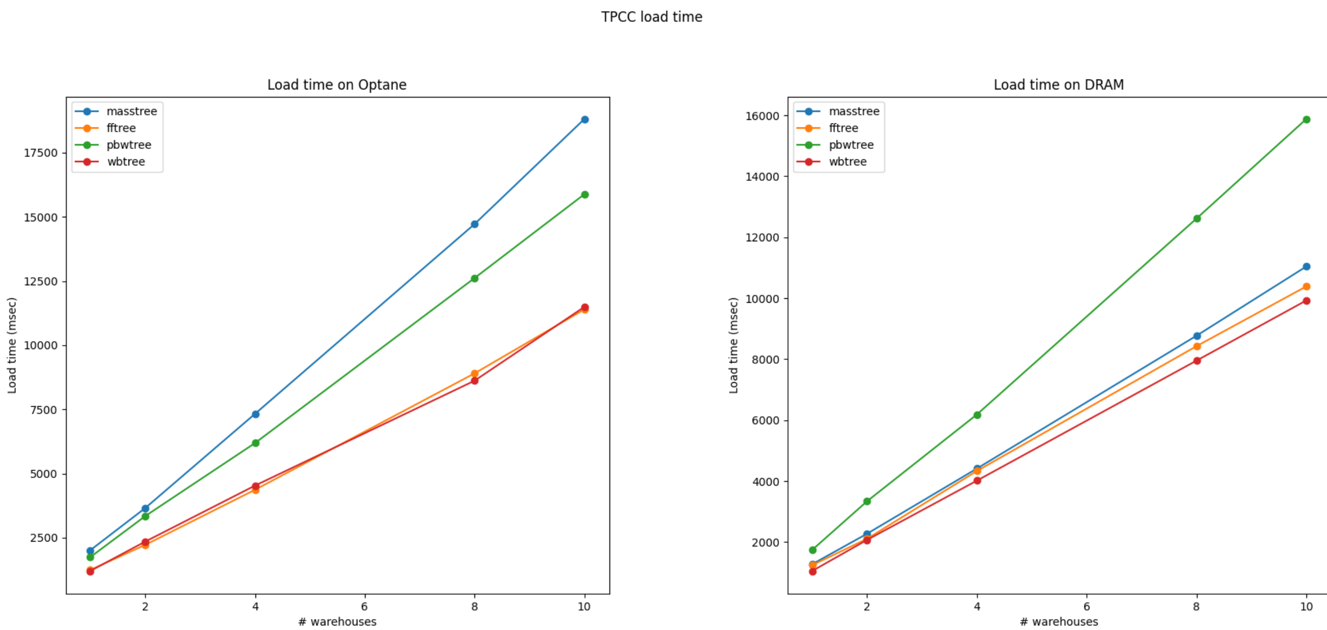


Figure 5.1: Comparative load times on Optane and DRAM

As mentioned, TPCC is mostly a write and update heavy workload. Its load phase performance is indicative of how an index performs under a write-heavy workload. The best

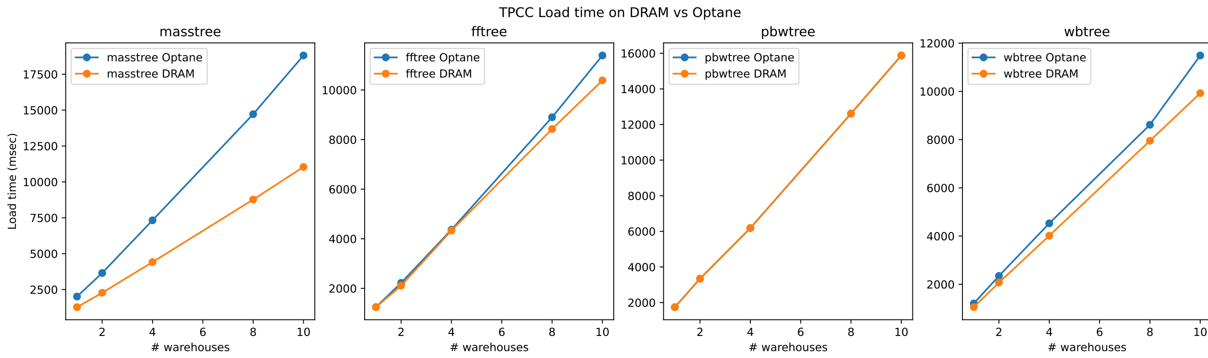


Figure 5.2: Per-index comparative load times on Optane and DRAM

performers in the load phase, requiring the least load time are Fast&Fair and WBtree which have almost identical performance (less than 1% difference). P-Masstree has the worst load performance and P-BwTree also does poorly. For 10 warehouses, P-Masstree is approximately 1.6 times slower to load about 5 million keys, while P-BwTree is 1.43 times slower than both Fast&Fair and WBtree.

5.3.2 Run phase

We have evaluated TPCC with all combinations of the following configurations: Number of warehouses in [1,2,4,8,10] and number of transactions in [10k, 25k, 50k, 75k, 100k, 250k, 500k, 750k, 1mil].

5.3.2.1 Runtime performance - throughput

Figure 5.3 shows the runtime throughput achieved by each index on Optane and DRAM respectively, for a single client submitting transactions, 10 warehouses and increasing number of transactions. We can see the number of transactions does not much affect performance and there is a clear difference between all indexes in terms of performance regardless of the number of transactions executed.

Therefore we choose to present results for larger numbers of transactions which better represent a real-world application.

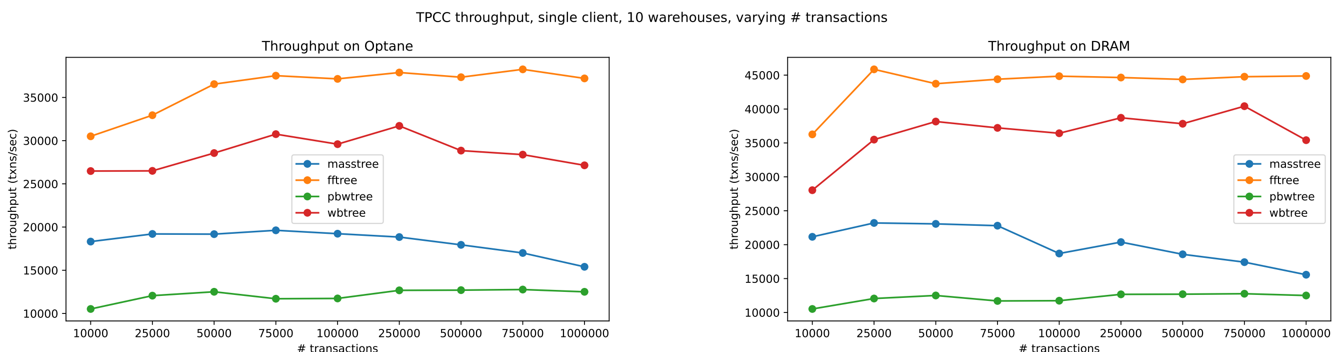


Figure 5.3: Comparative throughput on Optane and DRAM, 10 warehouses, single client

Figure 5.4 shows the runtime throughput achieved by each index on Optane and DRAM

respectively, for a single client submitting transactions, 750K transactions performed, for increasing number of warehouses.

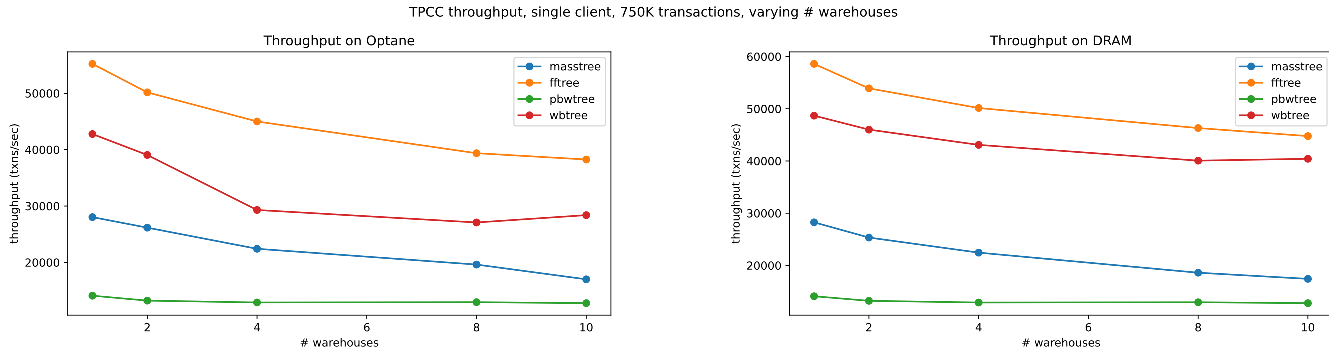


Figure 5.4: Comparative throughput on Optane and DRAM, 750k transactions, single client

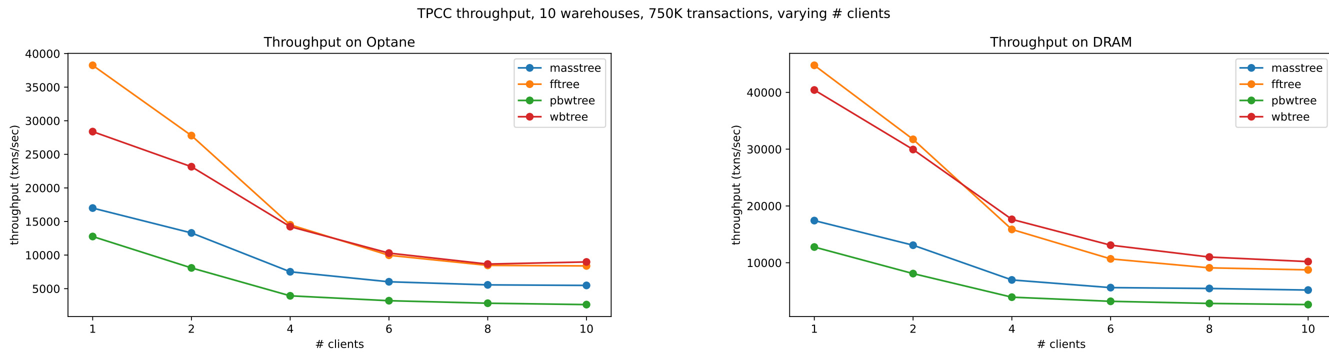


Figure 5.5: Comparative throughput on Optane and DRAM, 10 warehouses, 750K transactions, increasing number of clients

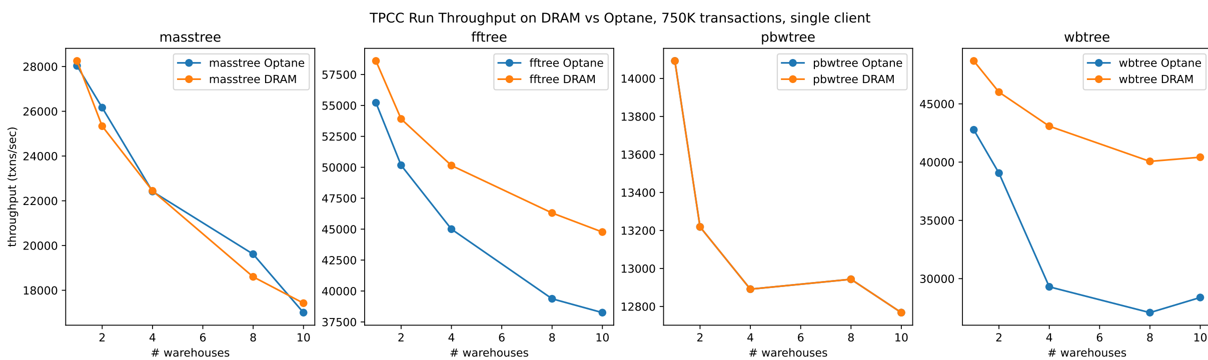


Figure 5.6: Comparative throughput on Optane and DRAM, 10 warehouses, 750K transactions, increasing number of warehouses

In the run phase, the best performing index in our TPCC experiments is Fast&Fair, followed by WBtree, P-Masstree and finally P-BwTree. Fast&Fair outperforms WBtree which is the next best performing index by achieving approximately 8.7% higher throughput and performing

8% better for a single warehouse, single client and 750 thousand transactions. It achieves 91% higher throughput than P-Masstree, performing approximately 47.6% better and 400% higher throughput than P-BwTree, performing 75.3% better for the same configuration. For 10 warehouses, Fast&Fair is 1.14x faster than wbtree, 2.24x faster than P-Masstree and 3.17x faster than P-BwTree.

Regarding the effect of multiple clients on runtime throughput, we observe the same trends in performance drop for all the indexes. Table 5.3 shows the runtime performance (measured in transactions per second) for 10 warehouses, 1 million transactions executed by a varying number of clients in the range 1 through 10. For more than 4 clients, transaction serialization and waiting on locks becomes the bottleneck, which is indicated by the much slighter difference in throughput.

Index	1 client	2 clients	4 clients	6 clients	8 clients	10 clients
Fast&Fair	37204.05	**26898.91**	**13560.13**	**9765.58**	**8940.94**	**8398.36**
WBtree	**37583.41**	20239.28	13281.38	9604.29	8689.69	8372.70
P-Masstree	15400.44	12070.3	7219.05	5766.27	5364.95	5212.51
P-BwTree	12195.86	7739.72	3978.07	3167.24	2911.19	2731.59
Percentage drop						
Fast&Fair		27.7%	49.6%	27.9%	8.4%	6%
WBtree		46.1%	34.3%	27.7%	9.5%	3.6%
P-Masstree		21.6%	40.1%	20.1%	6.9%	2.8%
P-BwTree		36.5%	48.6%	20.4%	8%	6.1%

Table 5.3: Effect of multiple clients on TPCC throughput

Increasing the number of clients does not allow us to evaluate the concurrency behavior of each index. This is additionally verified by including the WBtree, which is single-threaded, in experiments with multiple clients. This performance degradation, that we similarly observe for all indexes, is due to the transactional nature of the TPC-C benchmark, the isolation level we have implemented and the specifics of our implementation. As we are using shared mutexes to implement serializability with 2PL, but most operations modify several tables and only 8% of the transactions are read-only, this in practice leads to serial performance, with the additional significant overhead of lock contention. In order to evaluate the scalability of each index and its behavior in concurrent settings, we rely on diverse workloads of the YCSB benchmark. The behavior under YCSB is discussed later in this chapter.

5.3.2.2 Energy Consumption

Figure 5.7 shows the energy consumed by each index on Optane and DRAM respectively, for a single client submitting 750K transactions, and increasing number of warehouses. We observe that energy consumption is inversely proportional to throughput and execution time. The better an index performs, the less energy it consumes on both DRAM and Optane. Table 5.4 also summarizes the Optane energy consumption results for 1 and 10 warehouses for all indexes.

In terms of energy consumption, Fast&Fair consumes 5.7% less PMEM energy than WBtree, and also 52.1% less than P-Masstree and 58.4% less than P-BwTree. Masstree consumes 2.1x

more energy, P-BwTree consumes 2.4x more energy than FastFair.

Index	1 Warehouse	10 Warehouses
FastFair	2.32 kJ	4.80 kJ
WBtree	2.29 kJ	5.09 kJ
P-Masstree	4.14 kJ	10.05 kJ
P-BwTree	8.16 kJ	11.56 kJ

Table 5.4: Energy consumption for 1 warehouse, 10 warehouses

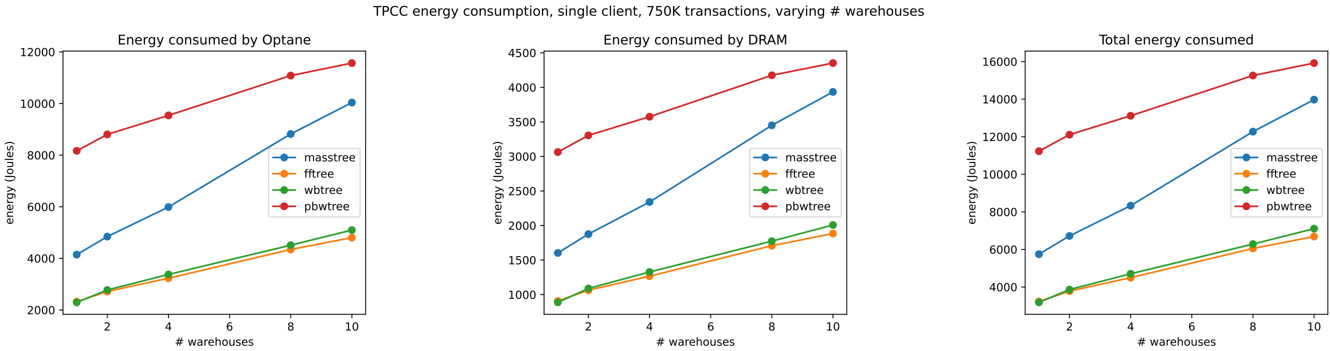


Figure 5.7: Consumed energy when running on Optane, 10 warehouses, 750K transactions, increasing number of clients

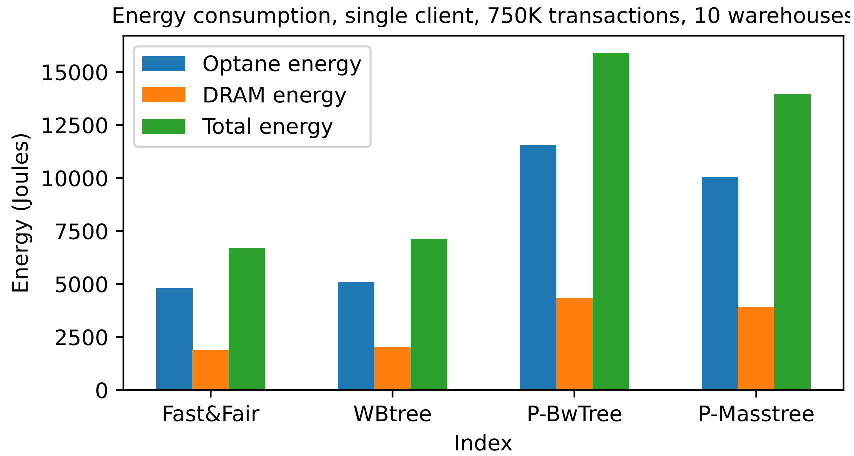


Figure 5.8: Consumed energy for 10 warehouses, 750K transactions, single client

5.3.2.3 Memory Accesses

With respect to memory accesses to Optane, we observe that they tend to increase with increasing number of clients, as well as with increasing number of warehouses. This observation is inline with the throughput drop we have observed as the number of clients increases and as the number of warehouses increases. We also observe that FastFair and WBtree, the two best performers, have a similar write access pattern and appear to make fewer write accesses to NVM.

The better performance achieved by FastFair and WBtree can be attributed to the fact that they minimize write operations to NVM; FastFair does so by removing the need for logging operations with its FAIR algorithm and WBtree by maintaining unsorted nodes and atomically updating a bitmap and indirection slot array instead.

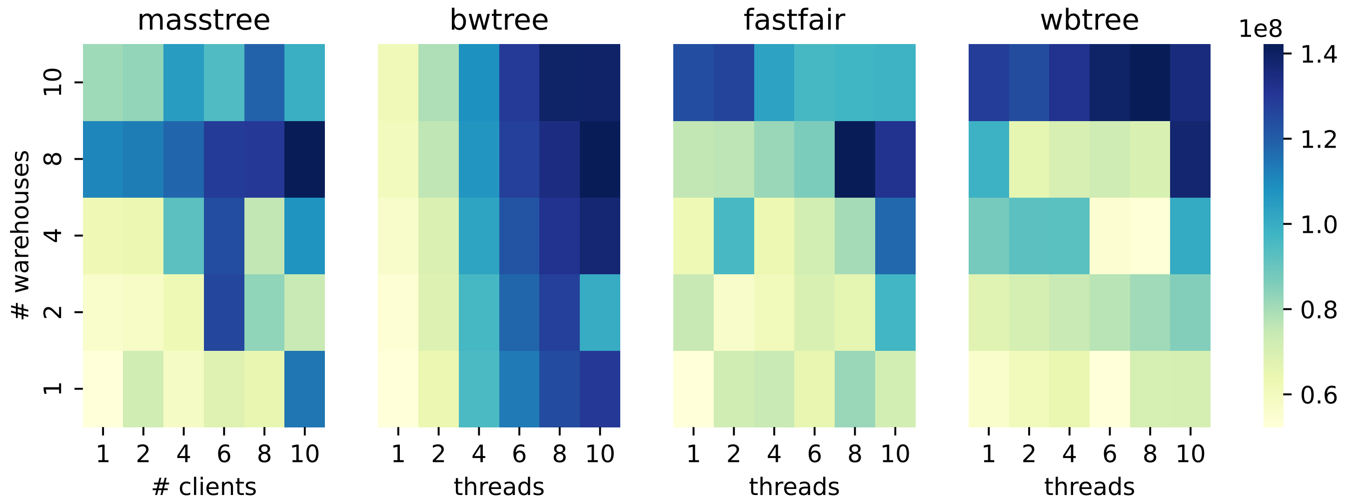


Figure 5.9: TPCC Optane write accesses, 750K transactions

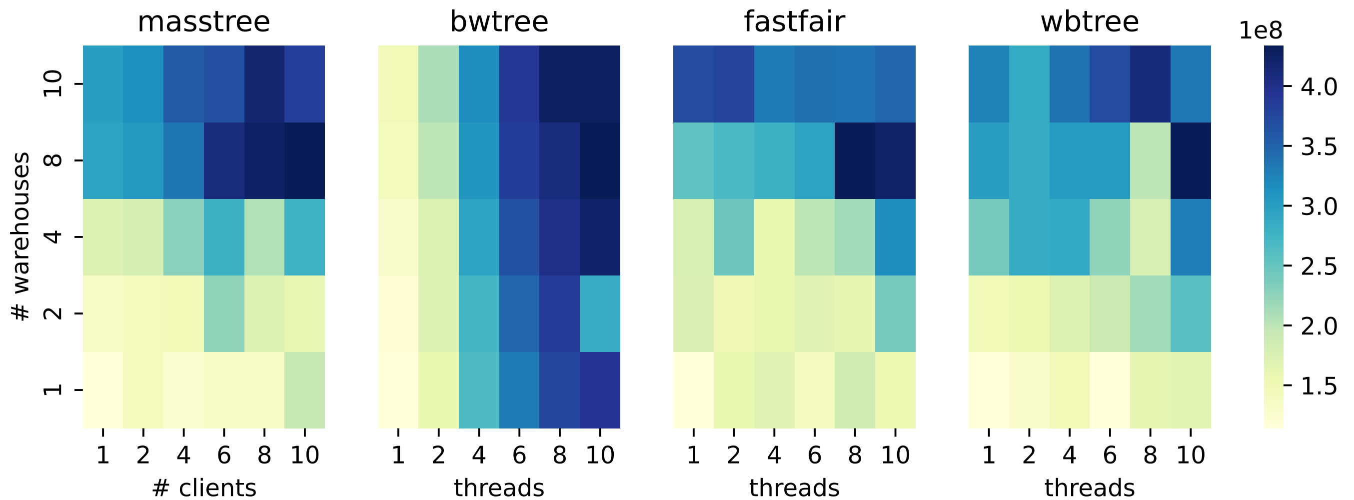


Figure 5.10: TPCC Optane read accesses, 750K transactions

5.4 YCSB (microbenchmark)

Yahoo Cloud Serving Benchmark[47] (or YCSB) is a popular key-value store benchmark framework. It is configurable and specifies different workloads, each representing a particular mix of read/write operations (insert, update, read, scan) , data sizes, request distributions and so on. allows selecting different key distributions. It also defines some standard workloads that emulate the specific cloud applications shown in Figure 5.11:

Workload	Operations	Record selection	Application example
A—Update heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B—Read heavy	Read: 95% Update: 5%	Zipfian	Photo tagging; add a tag is an update, but most operations are to read tags
C—Read only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (e.g., Hadoop)
D—Read latest	Read: 95% Insert: 5%	Latest	User status updates; people want to read the latest statuses
E—Short ranges	Scan: 95% Insert: 5%	Zipfian/Uniform*	Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id)

Figure 5.11: YCSB workloads. Adapted from [47]

We have run the YCSB benchmark with workloads A, B, C, E for all evaluated indexes, with 8-byte integer keys, and with 24-byte string keys, all uniformly distributed.

5.4.1 Integer keys

In Figures 5.12 through 5.15 we can see the comparative load and run throughput on DRAM vs Optane for each index and workload type.

5.4.1.1 Load Phase Throughput

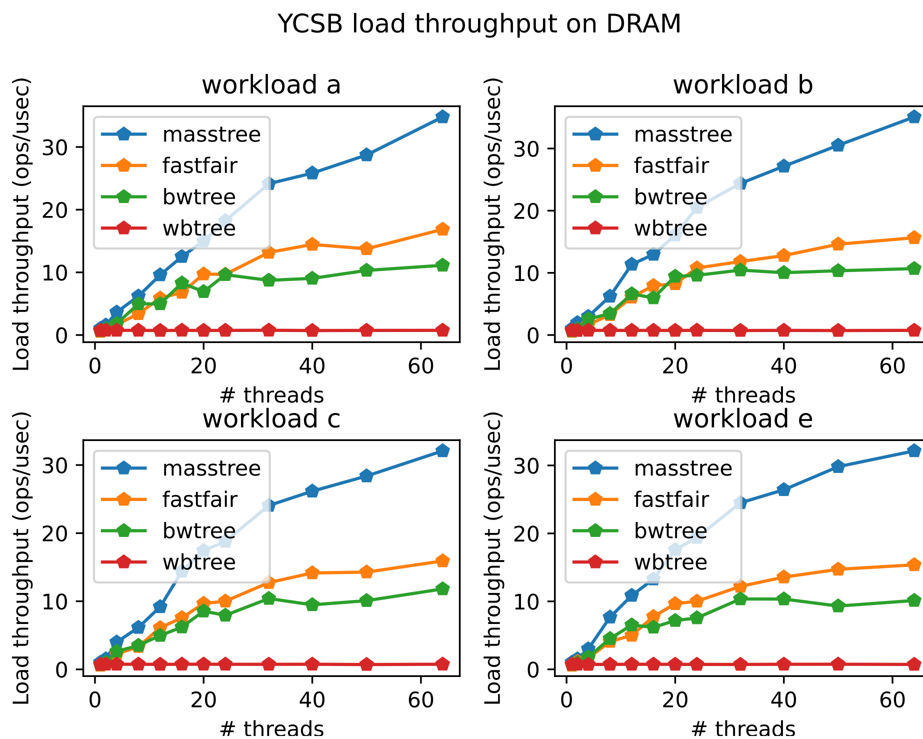


Figure 5.12: YCSB load throughput on DRAM for integer keys

In the load phase, BwTree appears to outperform all indexes for all workloads on Optane. Masstree does better than FastFair in terms of load throughput.

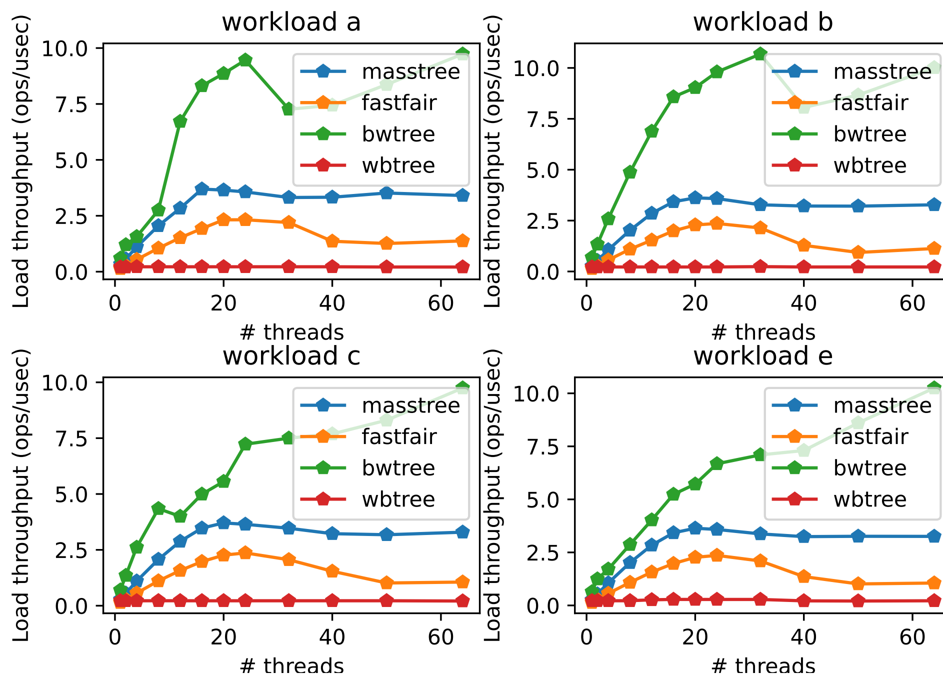


Figure 5.13: YCSB load throughput on Optane for integer keys

5.4.1.2 Run Phase Throughput

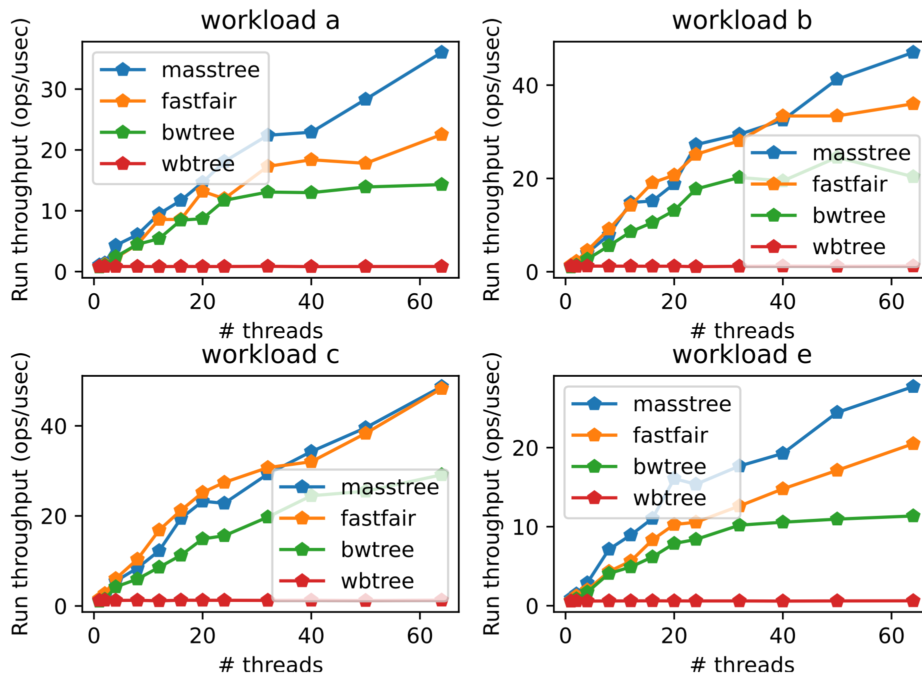


Figure 5.14: YCSB run throughput for integer keys on DRAM

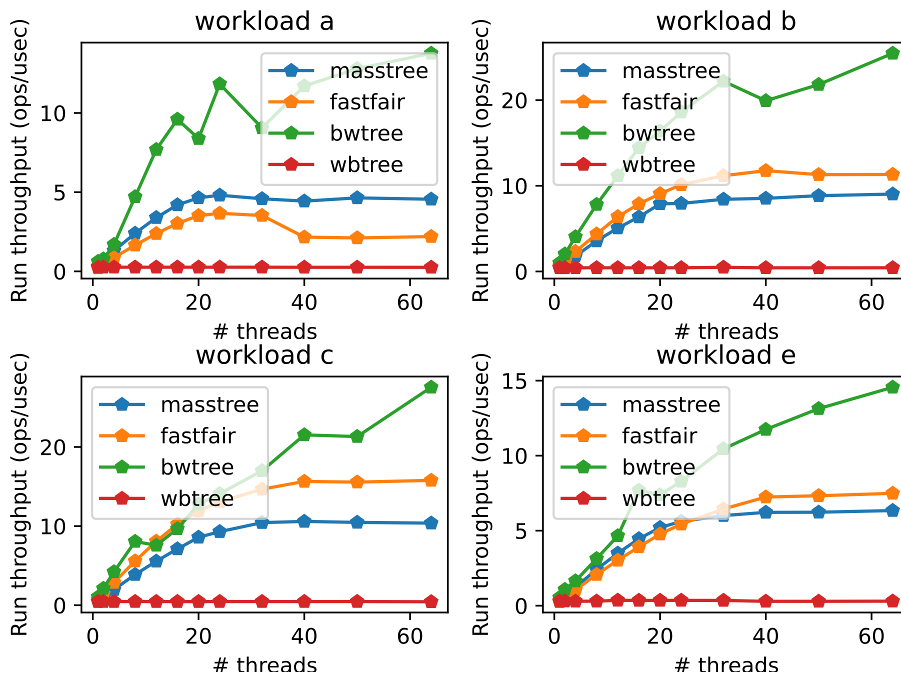


Figure 5.15: YCSB run throughput for integer keys on Optane

Regarding the performance on Optane, we find P-BwTree to have the best load perfor-

mance for all workloads. Masstree and FastFair perform close together. Masstree has better performance than FastFair for the load phase and for workload A, which is update-heavy, but is outperformed in the other workloads.

5.4.1.3 DRAM vs PMEM Comparison

This comparison allows us to observe how well indexes scale on Optane. We can see that Masstree and FastFair reach their peak at around 20 threads on Optane, whereas on DRAM they continue to scale well beyond that.

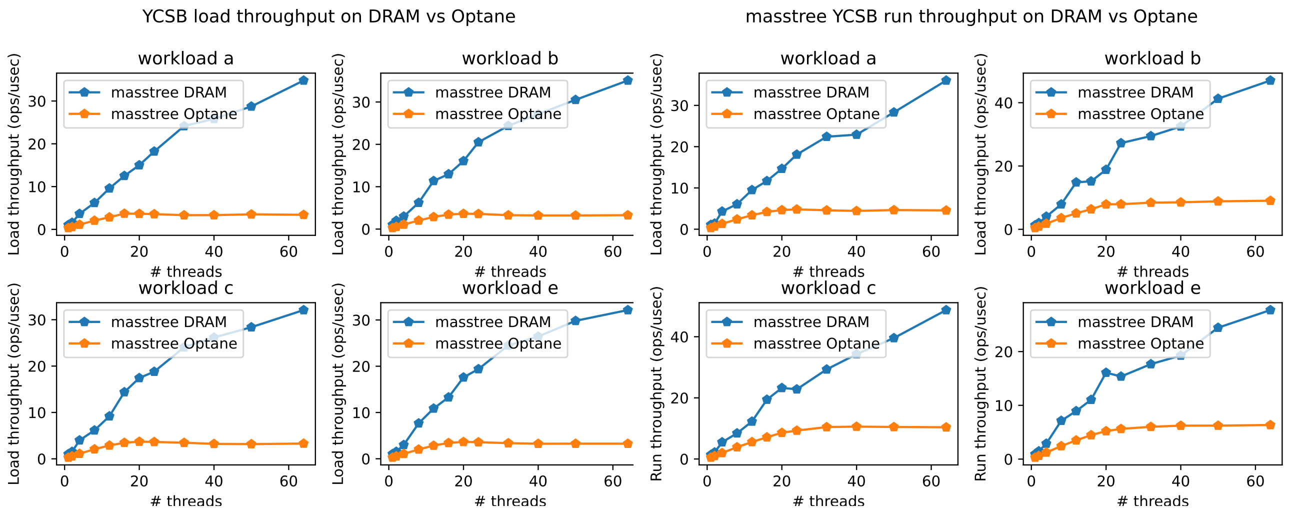


Figure 5.16: Masstre Comparative DRAM vs Optane throughput, integer keys

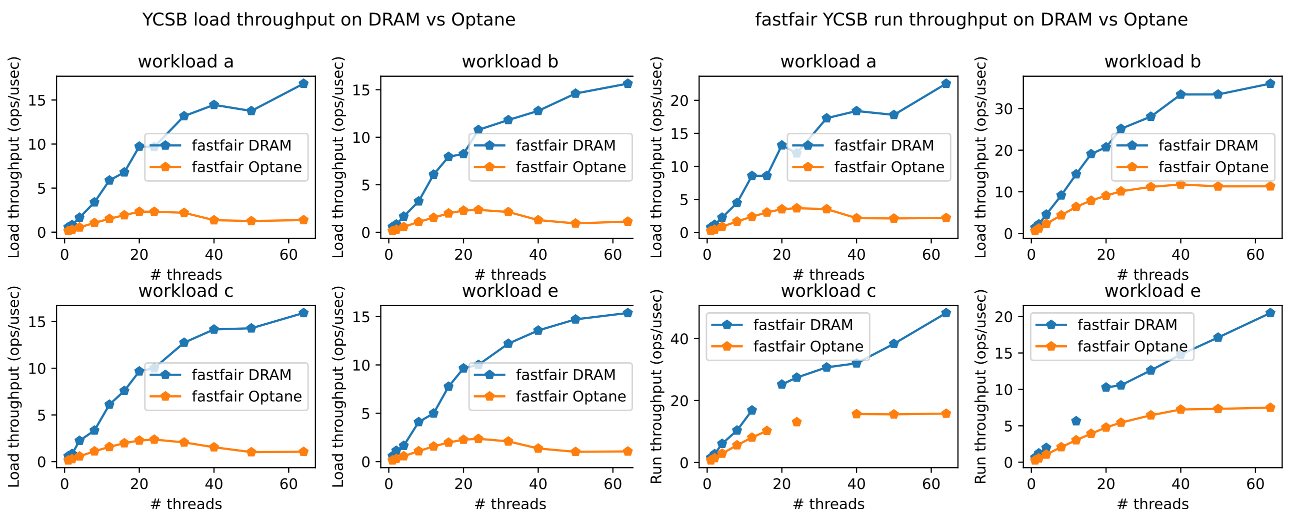


Figure 5.17: FastFair Comparative DRAM vs Optane throughput, integer keys

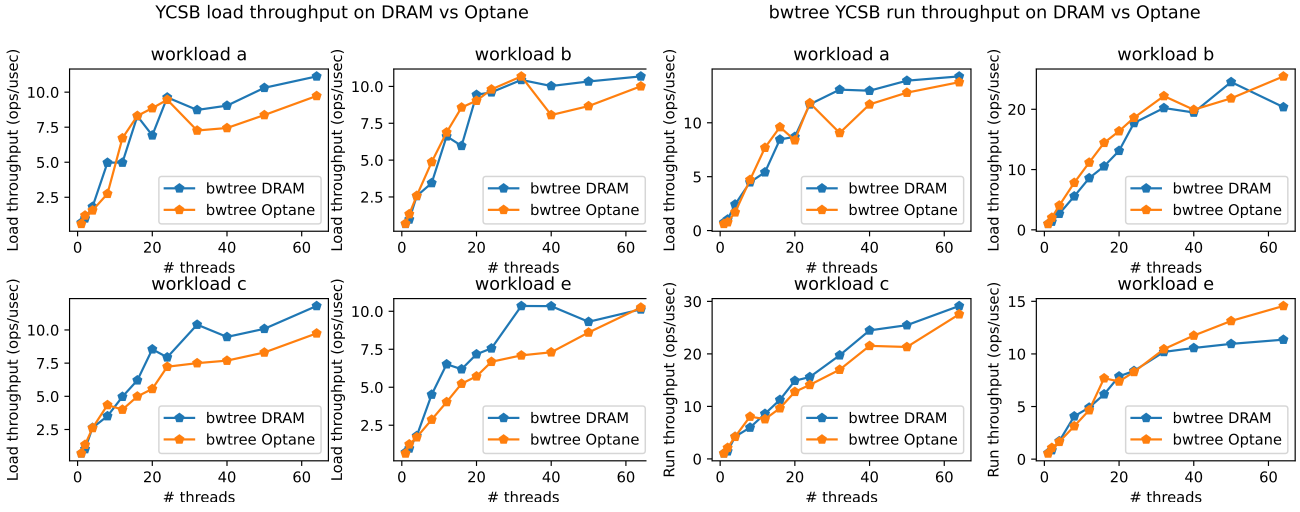


Figure 5.18: BwTree Comparative DRAM vs Optane throughput, integer keys

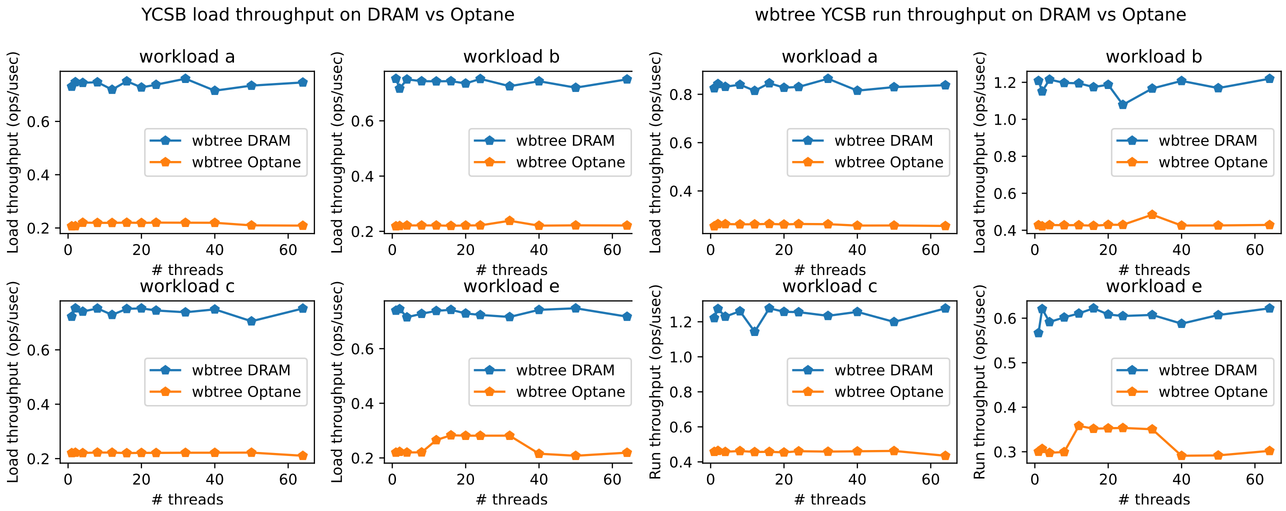


Figure 5.19: WBtree Comparative DRAM vs Optane throughput, integer keys

5.4.1.4 Energy Consumption

Energy is measured for the entire duration of the run, including the load and run phase, with a sampling rate of 0.01 sec for pcm-power. In Figures 5.20 through 5.23 we can see the energy consumption of each index. Again we observe the same trends as for the TPCC experiments: Indexes with better performance consume less energy. At 20 threads, where Masstree and FastFair stop scaling, energy consumption on NVM for each index and workload combination is shown in Table 5.5. We can see that while Masstree performs generally worse than FastFair, it consumes slightly less energy (from 5 up to 18% less, depending on the workload).

Index	A	B	C	E
FastFair	9.06 kJ	9.26 kJ	7.62 kJ	10.52 kJ
P-BwTree	7.12 kJ	5.8 kJ	6 kJ	7.27 kJ
P-Masstree	9.52 kJ	8.16 kJ	7.28 kJ	8.48 kJ
WBtree	87.68 kJ	55.97 kJ	60 kJ	67.82 kJ

Table 5.5: Energy consumption at 20 threads for integer keys

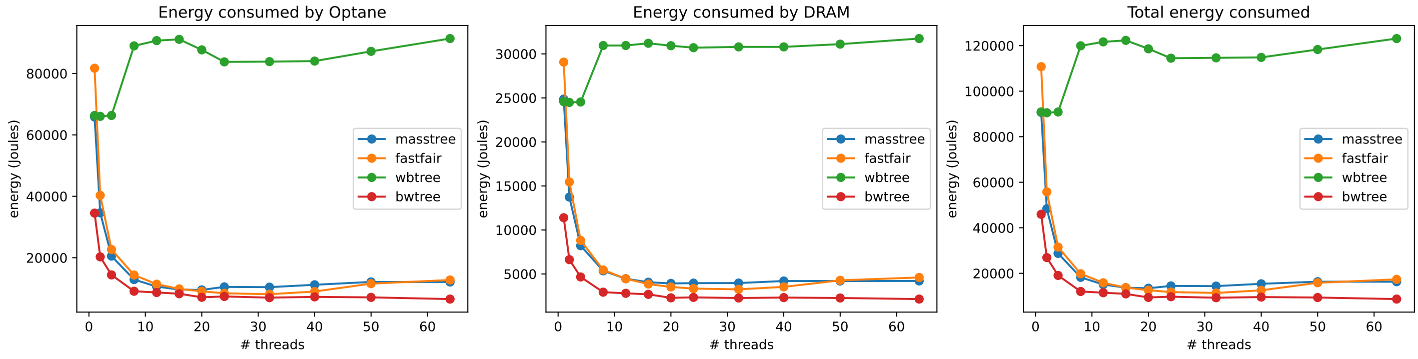


Figure 5.20: YCSB energy consumption on Optane for workload A

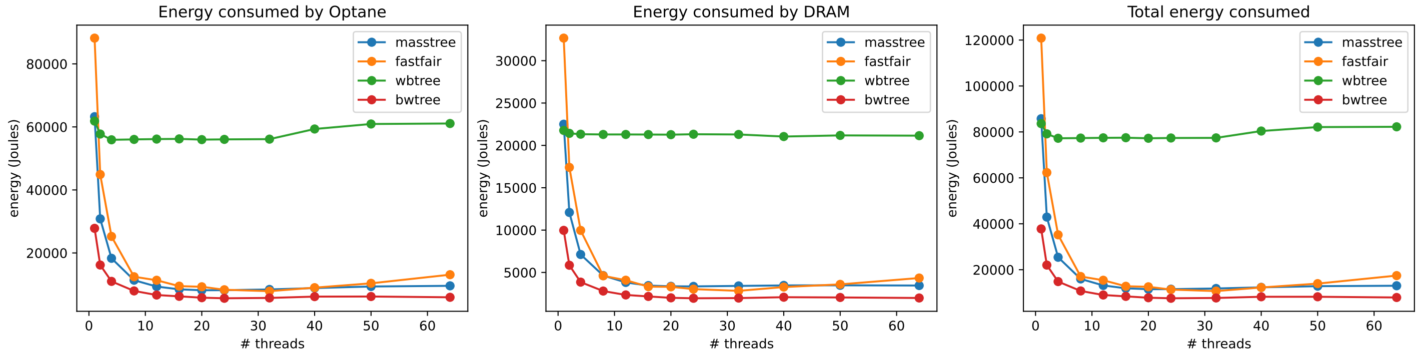


Figure 5.21: YCSB energy consumption on Optane for workload B

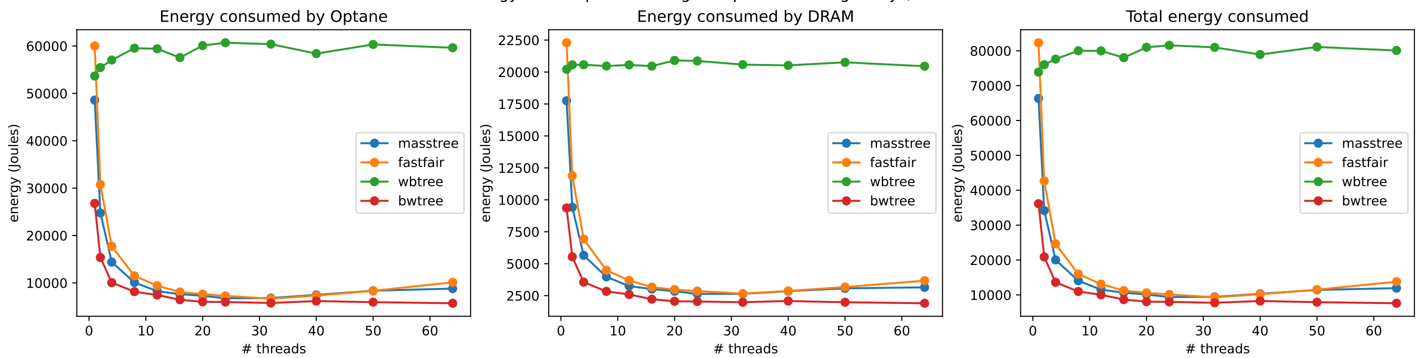


Figure 5.22: YCSB energy consumption on Optane for workload C

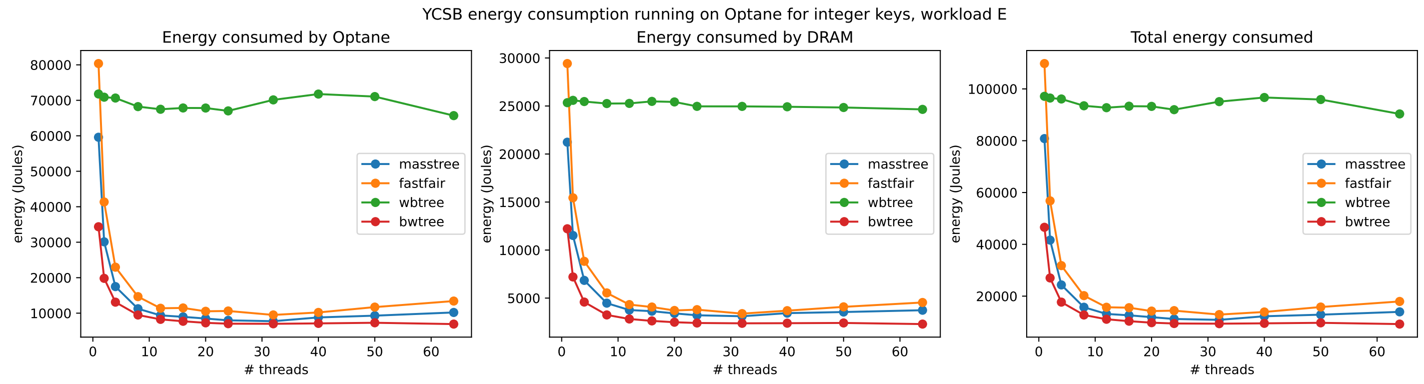


Figure 5.23: YCSB energy consumption on Optane for workload E

5.4.1.5 Memory Accesses

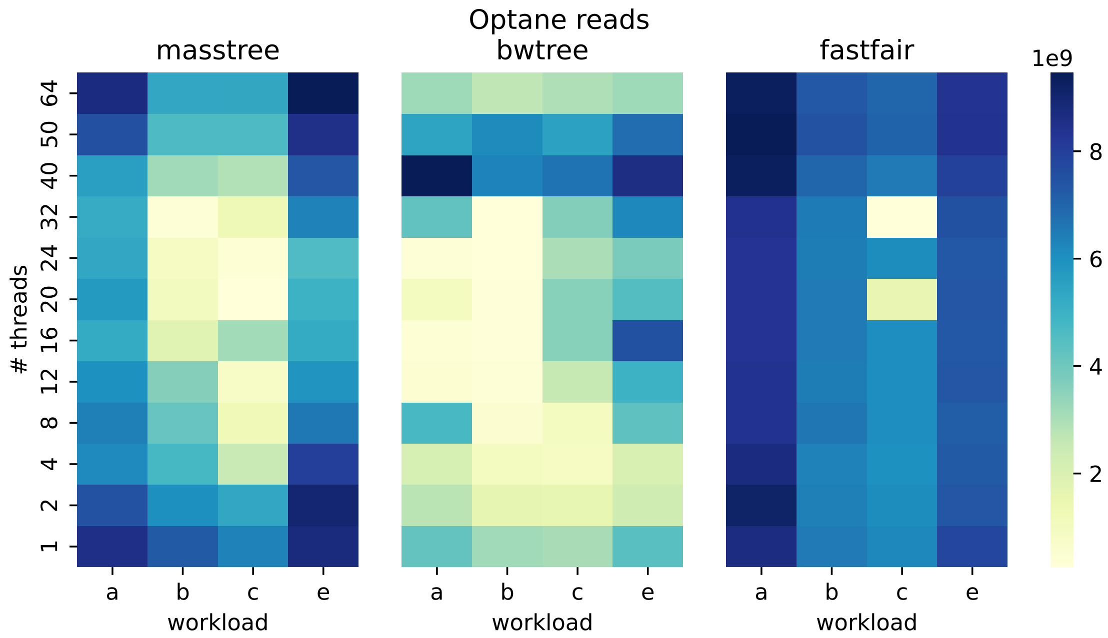


Figure 5.24: YCSB Optane reads, integer keys

Since WBtree is single-threaded, we present the memory accesses for each workload for a single thread in Table 5.6.

Workload	Reads	Writes
A	10320349444	2873135412
B	8218588896	1867941884
C	7851467292	1678547880
E	8922810740	1867332216

Table 5.6: Total Optane accesses for single-threaded wbtree, integer keys

For memory accesses, we observe that as the number of threads increases, so do accesses to

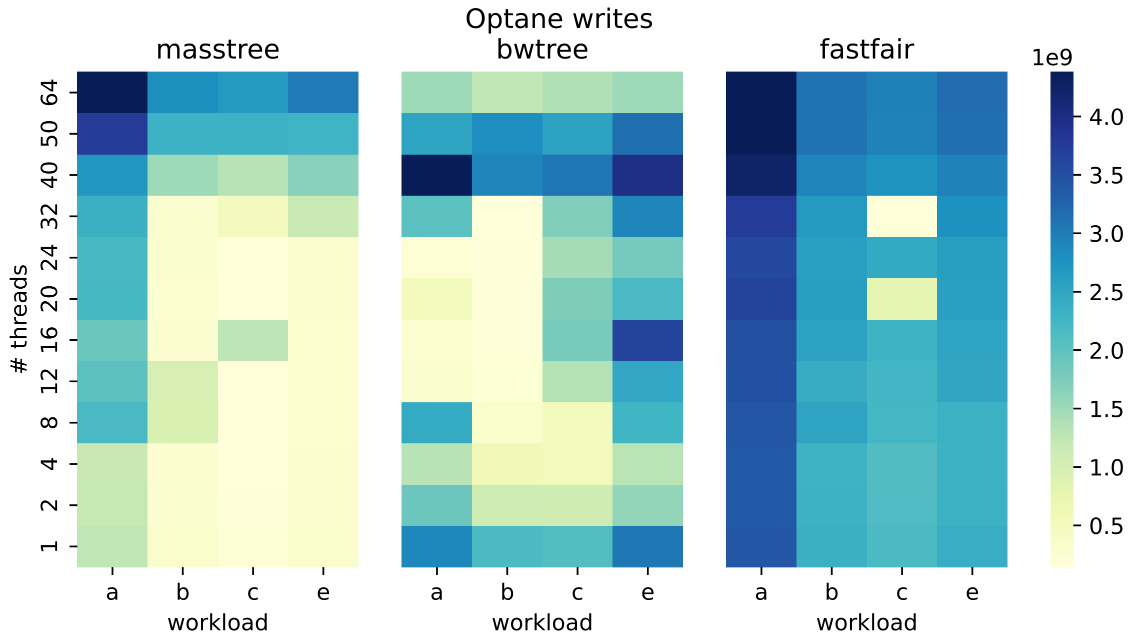


Figure 5.25: YCSB Optane writes, integer keys

NVM. We can also see that FastFair makes more accesses to NVM than Masstree does, which could explain the lower energy consumption by Masstree despite the lower performance.

5.4.2 String keys

WBtree implementation for string keys does not support the Range Scan operation, therefore we have excluded workload E from the experiments.

5.4.2.1 Throughput

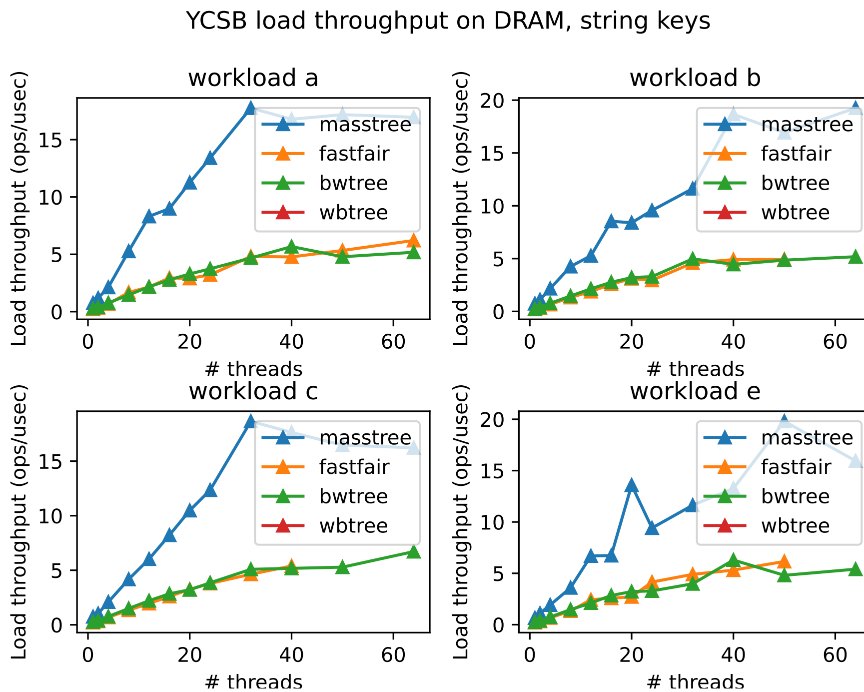


Figure 5.26: YCSB load throughput on DRAM for string keys

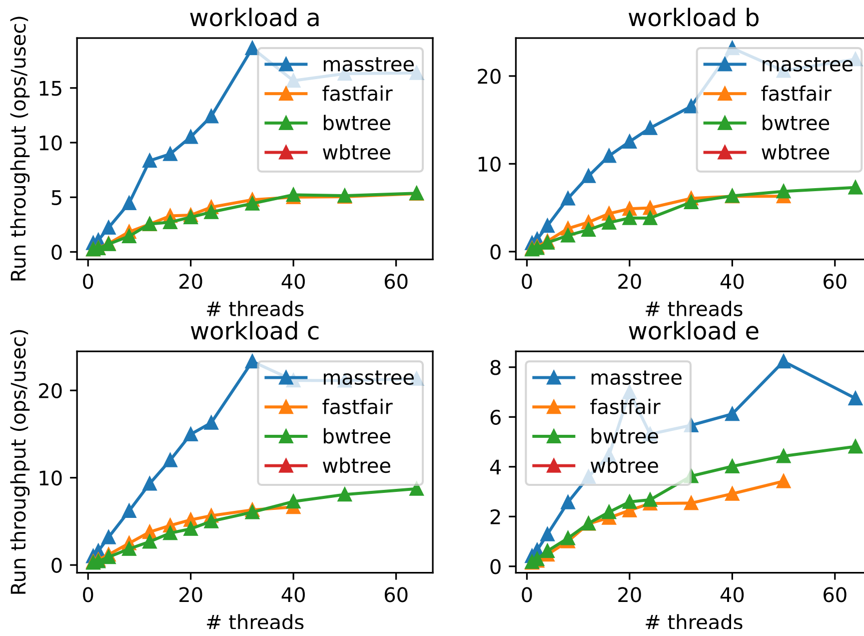


Figure 5.27: YCSB run throughput on DRAM for string keys

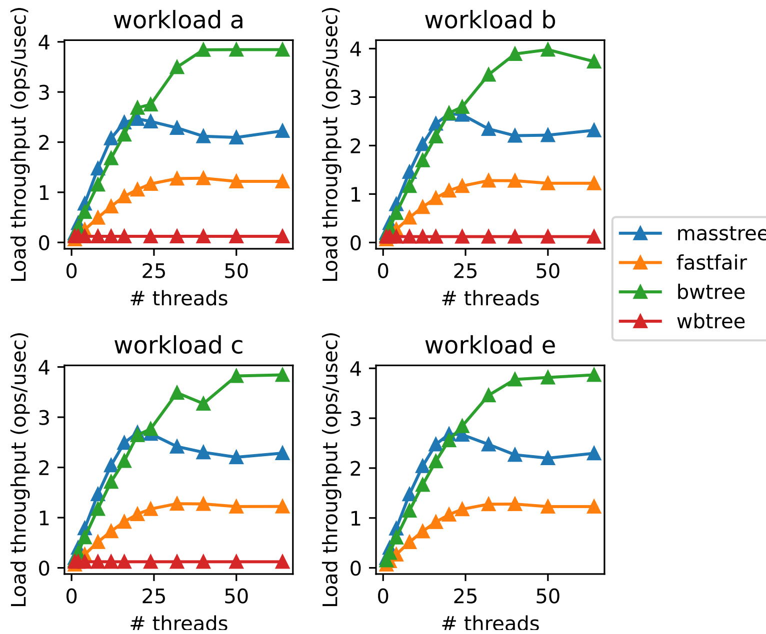


Figure 5.28: YCSB load throughput on Optane for string keys

The best performers for string keys are P-Masstree and P-BwTree. P-Masstree does better than P-BwTree in the run phase overall, but P-BwTree does better in the load phase. Another general observation across workloads is that P-BwTree appears to scale much more than other indexes in terms of performance. P-Masstree seems to scale up to 20 threads on Optane. Fast&Fair appears to scale up to 20 threads as well. For workload A (read/update, 50/50), run phase performance is dominated by P-Masstree until 20 threads. Then we observe that P-BwTree scales better with increasing number of threads. For workload B (read/write, 95/5), we observe that P-Masstree outperforms all other indexes, P-BwTree by achieving at least 1.2x better runtime performance at P-BwTree's peak throughput(50 threads), FastFair by 2.5x. The

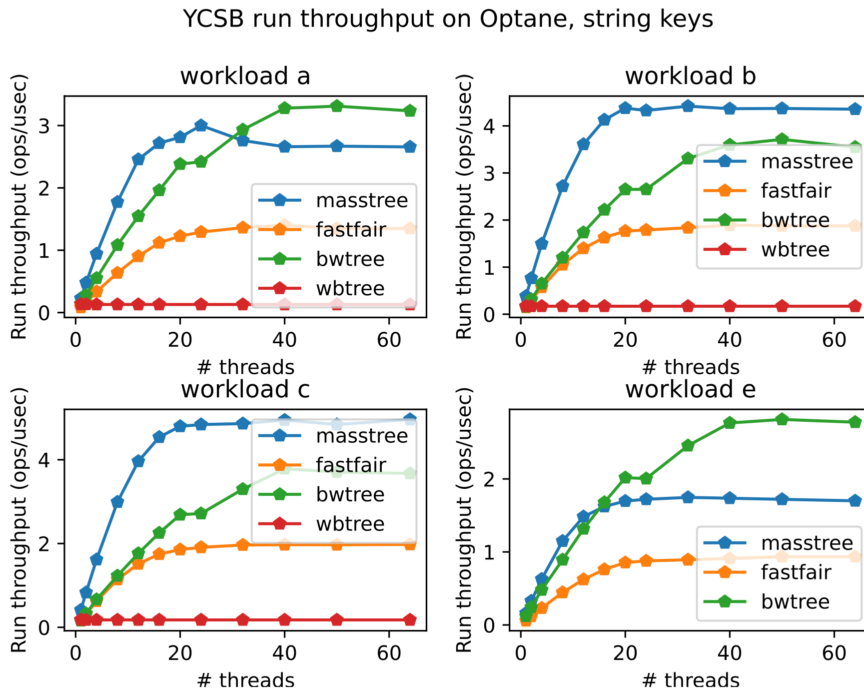


Figure 5.29: YCSB run throughput on Optane for string keys

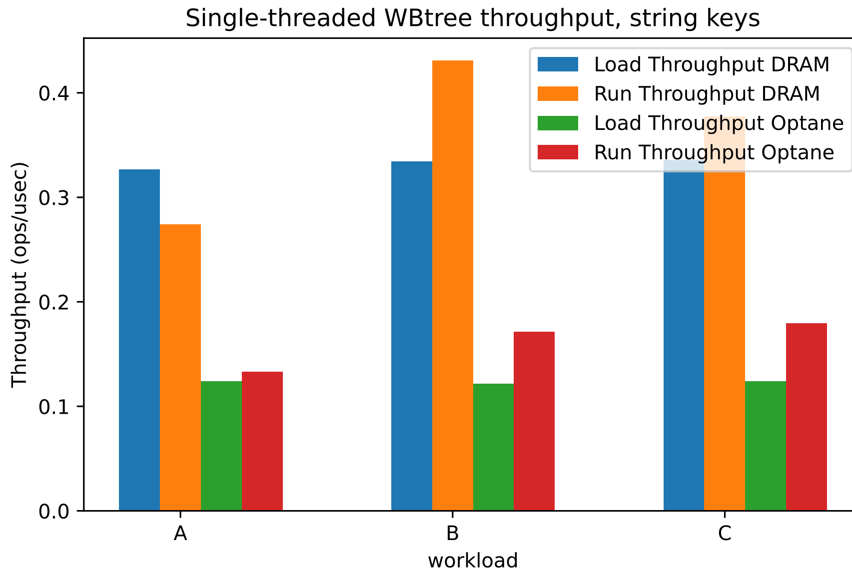


Figure 5.30: WBtree single-threaded YCSB throughput for string keys

relative behavior is the same for workload C. All indexes perform worse for workload E (scan-heavy) compared to their performance for other workloads. The difference is more pronounced for P-Masstree which is outperformed by P-BwTree. P-Masstree's good performance is expected due to its design that efficiently supports even variable-length string keys, by combining Trie and B+-tree architectures: a 24-byte string key is broken into 8-byte pieces that can use integer comparison to complete the operation much faster. P-BwTree also performs well for string keys, as it also has native support for string keys; FastFair and WBtree have to use pointers instead, which adds another level of indirection and negatively impacts performance.

At 20 threads, Masstree is 1.18x, 1.65x, 1.78x better in terms of operation throughput than BwTree and 2.3x, 2.48x, 2.58x better than FastFair for workloads A, B, C respectively.

5.4.2.2 Memory Accesses

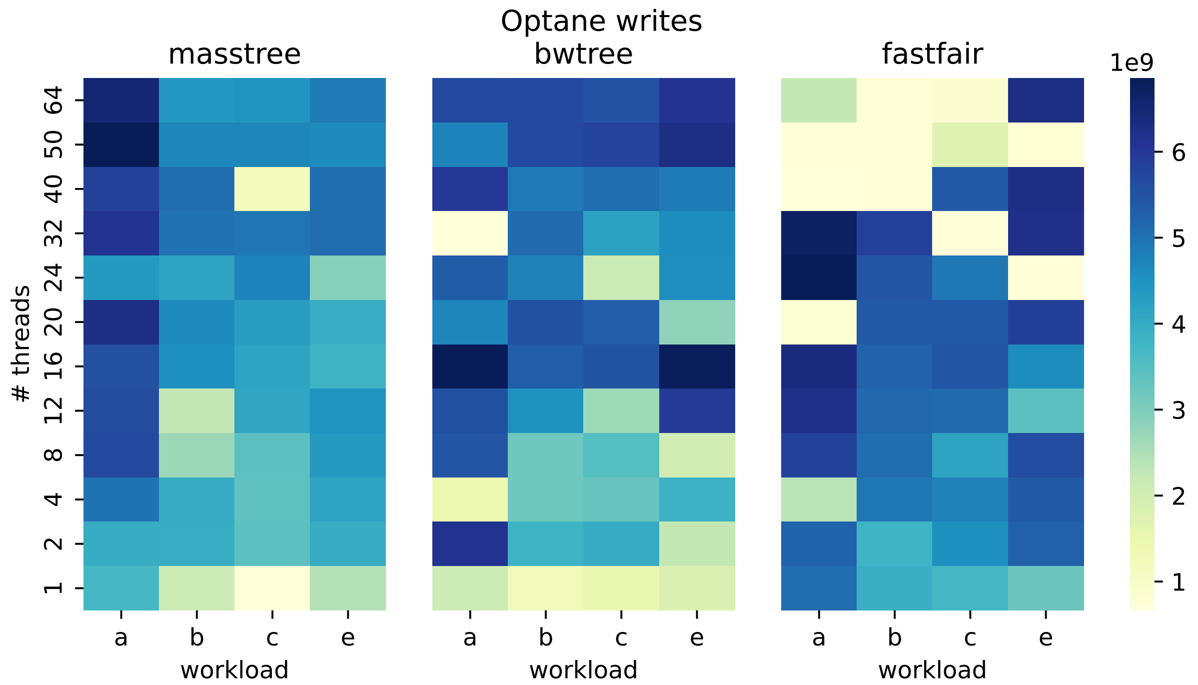


Figure 5.31: YCSB write accesses, string keys

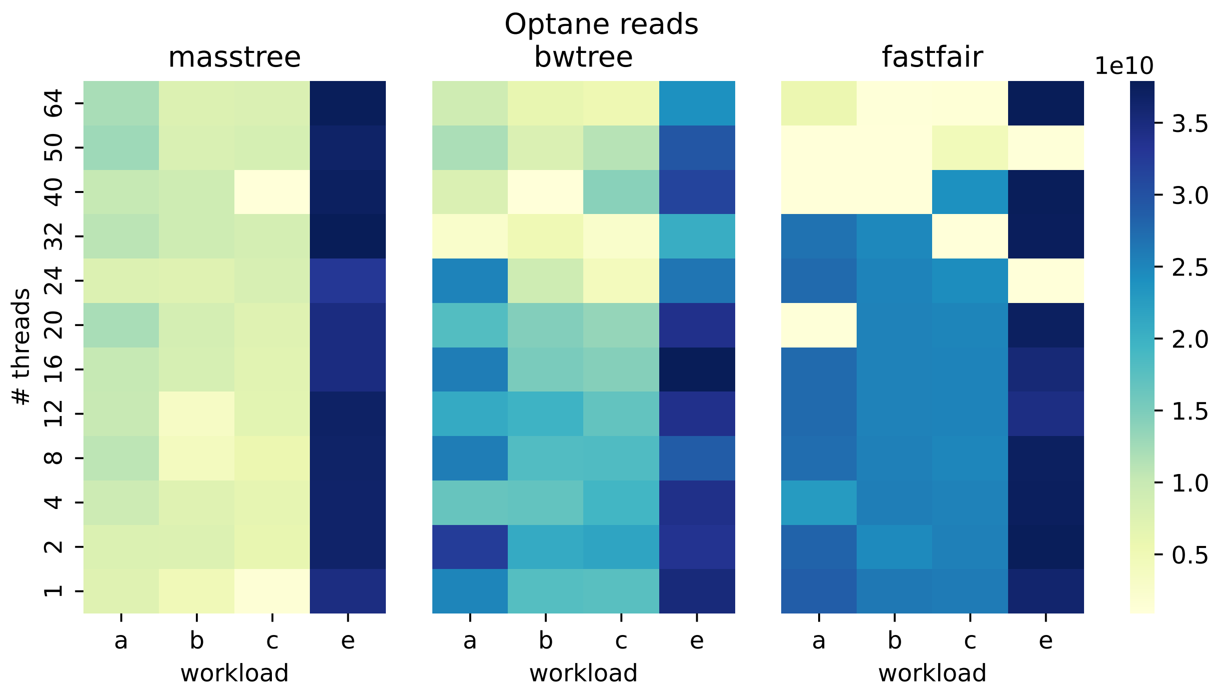


Figure 5.32: YCSB read accesses, string keys

For memory accesses, we again observe that as the number of threads increases, the accesses to NVM tend to increase as well.

Workload	Reads	Writes
A	25231356632	4545216304
B	23210235536	3353854820
C	22827446432	3165294056

Table 5.7: Total Optane accesses for single-threaded wbtree, string keys

5.4.2.3 Energy Consumption

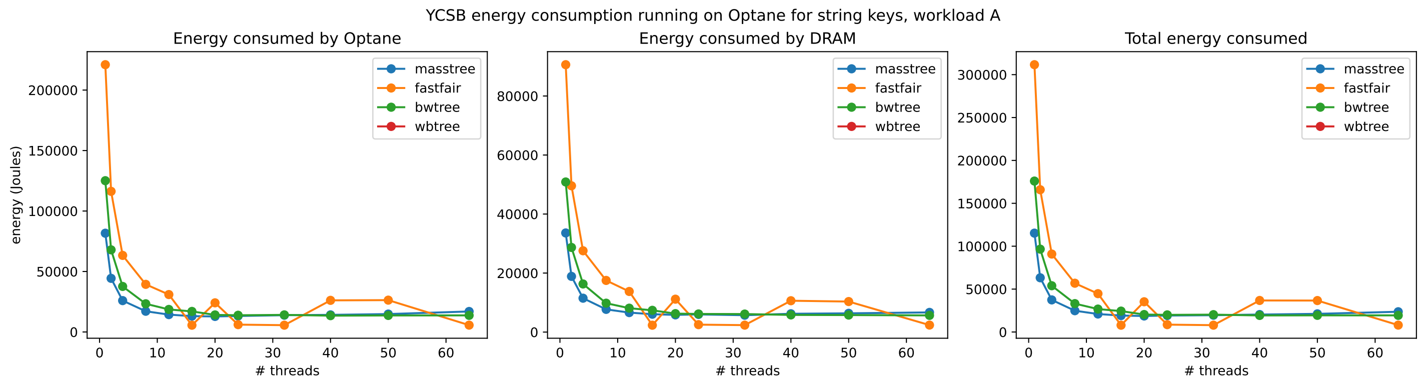


Figure 5.33: YCSB A energy consumption, string keys, running on Optane

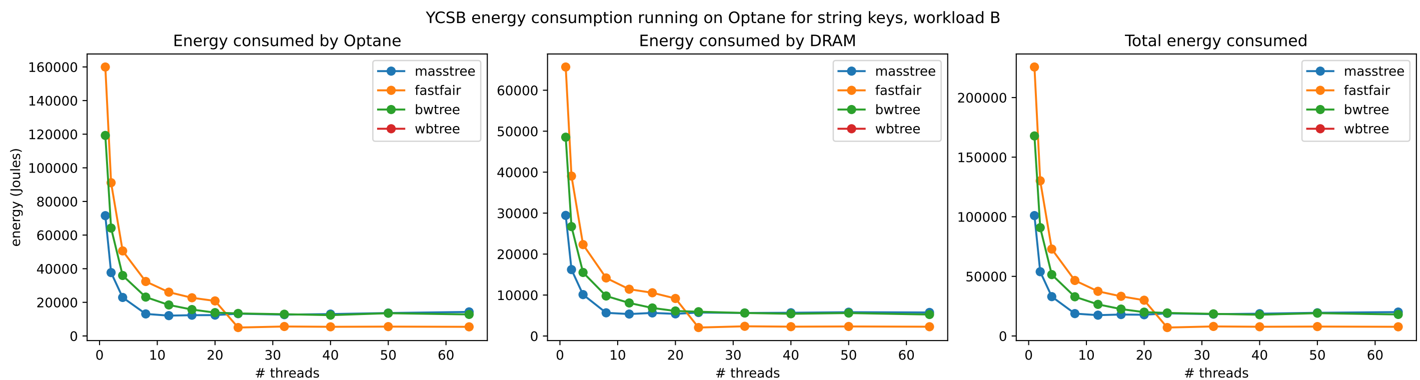


Figure 5.34: YCSB B energy consumption, string keys, running on Optane

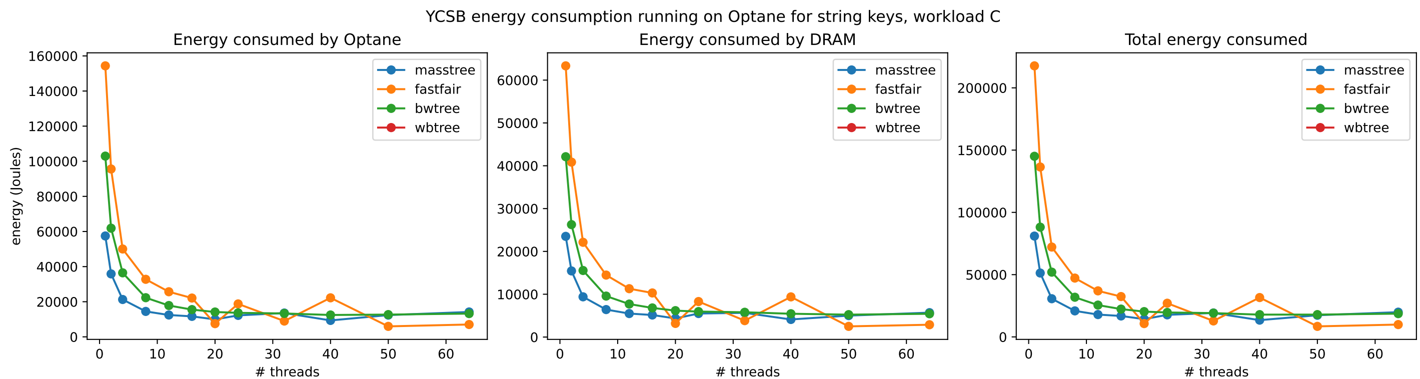


Figure 5.35: YCSB C energy consumption, string keys, running on Optane

Regarding energy consumption, we again observe that the higher the throughput, the lower the energy consumption.

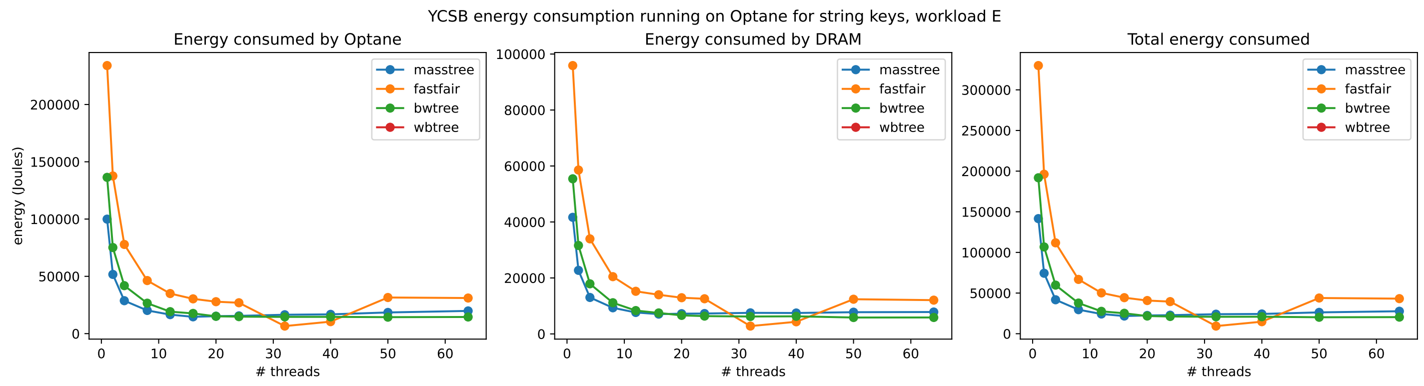


Figure 5.36: YCSB E energy consumption, string keys, running on Optane

5.4.3 Performance comparison of integer and string keys

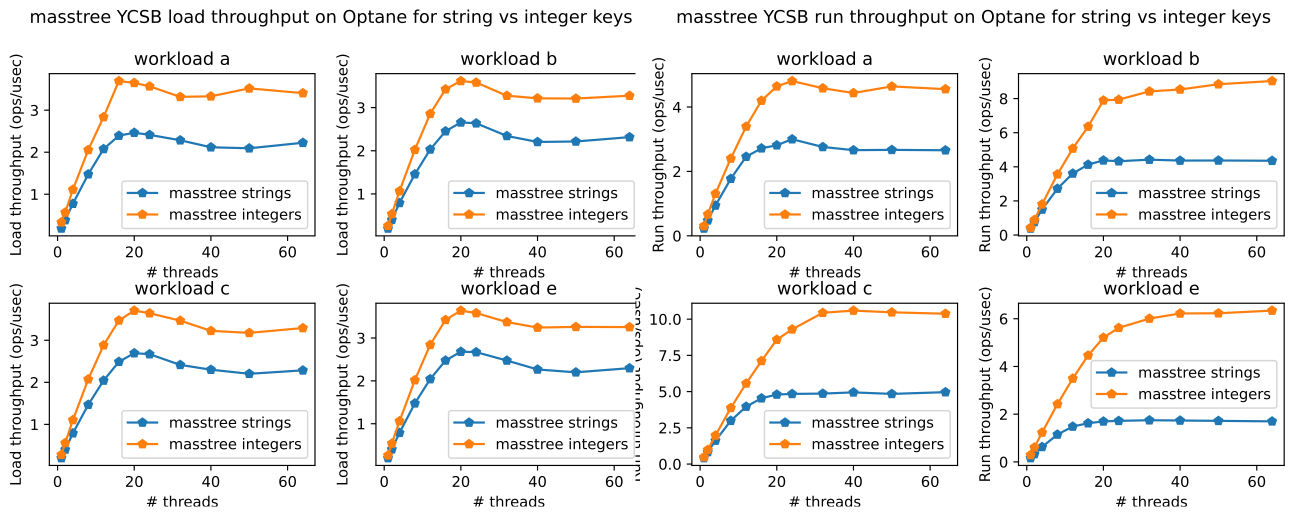


Figure 5.37: P-Masstree comparative performance for integer vs string keys. The figure on the left shows load throughput, the figure on the right shows run throughput for workloads A, B, C, E

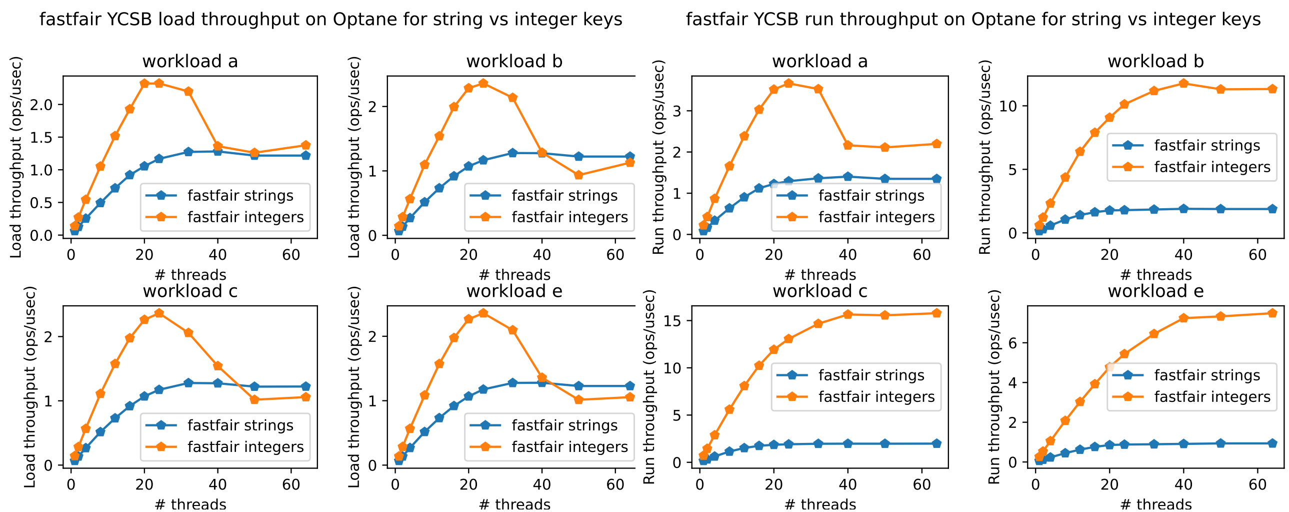


Figure 5.38: Fast&Fair comparative performance for integer vs string keys. The figure on the left shows load throughput, the figure on the right shows run throughput for workloads A, B, C, E

Throughput decreases significantly for all indexes for string keys compared to integer keys, as can be seen in Figures 5.37 through 5.40.

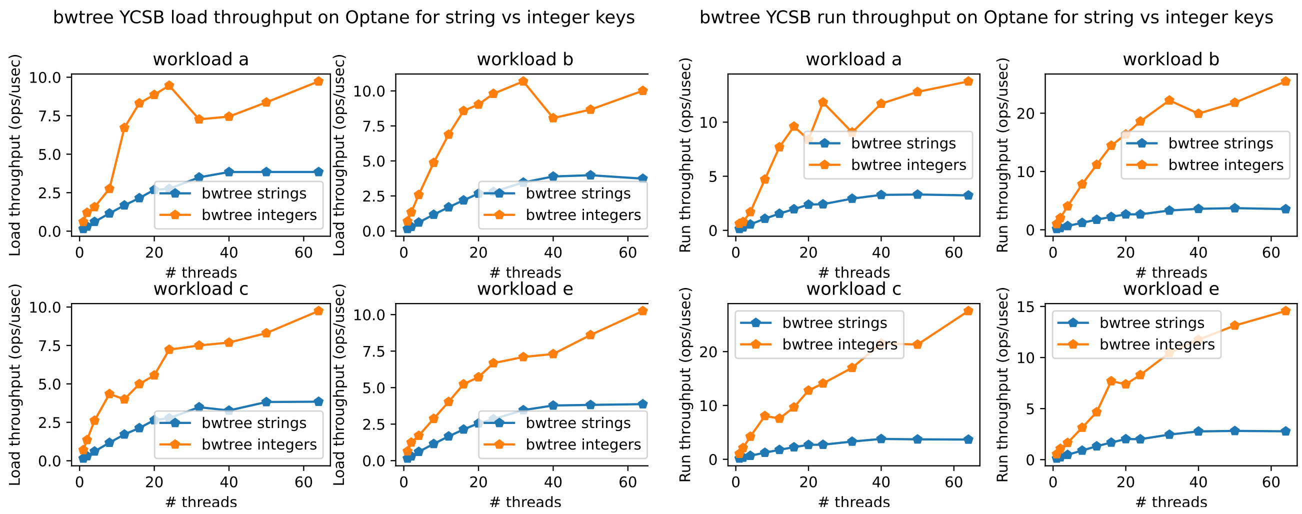


Figure 5.39: P-BwTree comparative performance for integer vs string keys. The figure on the left shows load throughput, the figure on the right shows run throughput for workloads A, B, C, E

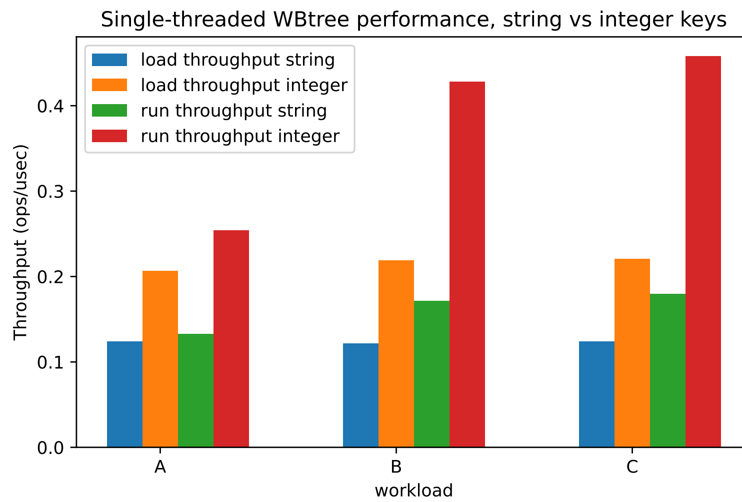


Figure 5.40: WBtree comparative performance for string vs integer keys

P-Masstree is the least impacted; performance drops for it about 39% at 20 threads for workload A(which is where it stops scaling) compared to integer keys, about 44% for workloads B and C and 67% for workload E. The performance trends however are similar to integer keys.

For P-BwTree, performance is at least 3.5 (workload A) and up to 6.5 times worse for string keys than it is for integer keys (workload B)

For Fast&Fair, performance is between 2.9(workload A) and 6.4x(workload C) worse for string keys.

For single-threaded WBtree, performance for string keys is almost 2x worse for workload A, 2.5x worse for B and C.

In terms of energy consumption compared to integer keys, indicatively for workload A we present the results in Table 5.8. Masstree consumes 1.3x more energy, FastFair 2.6x more energy and BwTree consumes 2x more energy compared to integer keys.

Index	**String**	**Integer**
Masstree	12.83 kJ	9.52 kJ
FastFair	24.1 kJ	9.06 kJ
BwTree	14.1 kJ	7.12 kJ

Table 5.8: Energy consumption at 20 threads, workload A, string vs integer keys

Chapter 6

Conclusions

6.1 Thesis Summary and Future Work

In this thesis we have evaluated different persistent B+-tree index structures, aiming to provide insights based on metrics such as operation throughput, energy consumption and scalability. We proposed an extendable evaluation methodology, that uses a transactional industry standard benchmark , the TPCC, in addition to the YCSB microbenchmark. Our evaluation shows that the best performing index under TPCC is FastFair; a close performer is WBtree; the converted Masstree and BwTree perform 3-4x worse. In terms of energy consumption, we find that the indexes which achieve the highest throughput also have the lowest energy consumption. Our experiments with the YCSB benchmark show that most indexes do not scale beyond 20 threads on Optane for any of the workloads, for both integer and string keys. Converted volatile indexes turn out to be very competitive with indexes tailored for NVM. We also observed that all indexes perform better for integer keys; even Masstree, which is designed to efficiently support arbitrary prefixes and variable length keys and outperforms the rest of the indexes for string keys, experiences at least a 39% performance drop compared to integer key performance.

In general, we conclude that there are tradeoffs:

- Different indexes turned out to be better performers for transactional workloads (represented by the TPCC benchmark) and for cloud applications (represented by the YCSB benchmark).
- String keys provide more flexibility, but they not only negatively impact performance but also increase energy consumption.

There are several ideas for extending the present work: A first extension idea concerns the TPCC benchmark implementation which could implement a less strict isolation level than serializability, since as we have seen in our experiments, throughput drops a lot with increasing number of clients, the main reason for that being lock contention. Additionally, our TPCC implementation is in-memory, and the scaling factor used and the number of executed transaction is relatively small. It would be interesting to employ an "industrial" version of TPCC, as is done for example in the work of [9], who have leveraged the custom storage engine feature of MySQL to evaluate their proposal's performance with TPCC, using it as a MySQL plugin.

Our evaluation is additionally limited with respect to the number of indexes included; there are recent proposals in the literature which have not been evaluated against the indexes we examined and their inclusion would be a useful extension of the present work. We could also

consider more low-level metrics for the evaluation, such as the number of cache line flush/sfence operations, cache misses etc.

Finally, our evaluation methodology could serve as a first step in developing a methodology for automatically selecting the appropriate index structure for a heterogeneous DRAM/NVM system, depending on the workload type and metrics to optimize for.

More in general, the following areas remain relatively unexplored in the existing literature:

- Impact of eADR (extended ADR). The impact that eADR could have on performance is unknown. eADR guarantees persistence of writes when they reach the CPU caches, as opposed to ADR which does not include them and guarantees persistence of the WPQ. eADR enables the use of optimizations such as Intel's TSX (transactional synchronization extensions), [53]. [2] also leave it to future evaluations to assess the impact, but they do not believe it will greatly impact their own conclusions. Authors of [1], have evaluated eADR by emulation, and they find that it improves throughput by less than a factor of 2, and think that the main benefit is "probably in terms of simplifying the programming model, leading to fewer bugs and saving in development and maintenance costs".

- Write endurance of PMEM [48]

Bibliography

[1] L. Lersch *et al.*, "Evaluating persistent memory range indexes," vol. 13. VLDB Endowment, Dec. 2019, pp. 574–587.

[2] Y. He *et al.*, "Evaluating persistent memory range indexes: Part two," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2477–2490, Jul. 2022.

[3] D. Hu *et al.*, "Persistent memory hash indexes: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 14, pp. 785–798, 2021.

[4] S. K. Lee *et al.*, "Recipe: Converting concurrent dram indexes to persistent-memory indexes." Association for Computing Machinery, Inc, Oct. 2019, pp. 462–477.

[5] M. Friedman *et al.*, "Nvtraverse: In NVRAM data structures, the destination is more important than the journey," *CoRR*, vol. abs/2004.02841, 2020. [Online]. Available: https://arxiv.org/abs/2004.02841

[6] A. Memaripour *et al.*, "Pronto," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, Mar. 2020.

[7] R. M. Krishnan *et al.*, *Tips: Making Volatile Index Structures Persistent with DRAM-NVMM Tiering*. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/krishnan

[8] In-memory tpc-c implementation. [Online]. Available: https://github.com/evanj/tpccbench

[9] B. Yan *et al.*, "Revisiting the design of LSM-tree based OLTP storage engine with persistent memory," *Proceedings VLDB Endowment*, vol. 14, no. 10, pp. 1872–1885, Jun. 2021.

[10] J. Arulraj, "The design and implementation of a non-volatile memory database management system," PhD dissertation, Carnegie Mellon University, 2018.

[11] S. Scargall, *Programming Persistent Memory*. Apress, 2020. [Online]. Available: https://doi.org/10.1007/978-1-4842-4932-1

[12] H.-K. Liu *et al.*, "A survey of non-volatile main memory technologies: State-of-the-arts, practices, and future directions," *Journal of Computer Science and Technology*, vol. 36, no. 1, pp. 4–32, Jan. 2021.

[13] Intel, "Intel® Optane™ persistent memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, [Online; Accessed 1 March 2022].

[14] D. Waddington *et al.*, "Evaluating intel 3d-xpoint NVDIMM persistent memory in the context of a key-value store," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, Aug. 2020.

[15] J. Yang *et al.*, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20).* Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182.

[16] Z. Wang *et al.*, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, Oct. 2020.

[17] W. Zhang *et al.*, "Chameleondb: a key-value store for optane persistent memory," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021.* ACM, 2021, pp. 194–209.

[18] J. Izraelevitz *et al.*, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019.

[19] L. Benson *et al.*, "Viper: An efficient hybrid pmem-dram key-value store," *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.

[20] A. van Renen *et al.*, "Building blocks for persistent memory: How to get the most out of your new memory?" vol. 29. Springer Science and Business Media Deutschland GmbH, 11 2020, pp. 1223–1241.

[21] P. Götze *et al.*, "Data management on non-volatile memory: A perspective," *Datenbank-Spektrum*, vol. 18, no. 3, pp. 171–182, Oct. 2018.

[22] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[23] J. Izraelevitz *et al.*, "Linearizability of persistent memory objects under a full-system-crash failure model," in *Lecture Notes in Computer Science*, ser. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327.

[24] P. L. Lehman and s. B. Yao, "Efficient locking for concurrent operations on b-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, p. 650–670, Dec. 1981.

[25] A. Silberschatz *et al.*, *ISE Database System Concepts*, 7th ed. Columbus, OH: McGraw-Hill Education, Mar. 2019.

[26] P. O'Neil *et al.*, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun. 1996.

[27] Google leveldb. [Online]. Available: https://github.com/google/leveldb

[28] C. Luo and M. J. Carey, "Lsm-based storage techniques: A survey," *The VLDB Journal*, vol. 29, no. 1, p. 393–418, Jul. 2019.

[29] Facebook rocksdb. [Online]. Available: https://github.com/facebook/rocksdb

[30] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990.

[31] A. Petrov, *Database Internals: A deep dive into how distributed systems work*. O'Reilly Media, 2019.

[32] A. Shanbhag *et al.*, "Large-scale in-memory analytics on intel® optanetm dc persistent memory," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, ser. DaMoN '20, 2020.

[33] A. V. Renen *et al.*, "Managing non-volatile memory in database systems." Association for Computing Machinery, May 2018, pp. 1541–1555.

[34] J. Arulraj and A. Pavlo, "How to build a non-volatile memory database management system," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, May 2017.

[35] C. Chen *et al.*, "Optimizing in-memory database engine for AI-powered on-line decision augmentation using persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 799–812, Jan. 2021.

[36] X. Zhou *et al.*, "Dptree: Differential indexing for persistent memory," vol. 13. VLDB Endowment, 12 2019, pp. 421–434.

[37] W. H. Kim *et al.*, "Pactree: A high performance persistent range index using pac guidelines," *SOSP 2021 - Proceedings of the 28th ACM Symposium on Operating Systems Principles*, pp. 424–439, 2021.

[38] J. Arulraj *et al.*, "Bztree," *Proceedings VLDB Endowment*, vol. 11, no. 5, pp. 553–565, Jan. 2018.

[39] Y. Mao *et al.*, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*. ACM Press, 2012.

[40] D. Hwang *et al.*, "Endurable transient inconsistency in Byte-Addressable persistent B+-Tree," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 187–200.

[41] J. J. Levandoski *et al.*, "The bw-tree: A b-tree for new hardware platforms," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2013.

[42] Z. Xie *et al.*, "A comprehensive performance evaluation of modern in-memory indices," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2018. [Online]. Available: https://doi.org/10.1109/icde.2018.00064

[43] S. Chen and Q. Jin, "Persistent B $^{+}$ -trees in non-volatile main memory," *Proceedings VLDB Endowment*, vol. 8, no. 7, pp. 786–797, Feb. 2015.

[44] V. Leis *et al.*, "The ART of practical synchronization," in *Proceedings of the 12th International Workshop on Data Management on New Hardware*. ACM, Jun. 2016.

[45] Pibench. [Online]. Available: https://github.com/sfu-dis/pibench

[46] Tpc-c. [Online]. Available: https://www.tpc.org/tpcc/

[47] B. F. Cooper *et al.*, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*. ACM Press, 2010.

[48] M. Katsaragakis *et al.*, "Energy consumption evaluation of optane dc persistent memory for indexing data structures," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2022, pp. 75–84.

[49] Intel performance counter monitor (pcm). [Online]. Available: https://github.com/intel/pcm

[50] M. Katsaragakis *et al.*, "Adjacent LSTM-Based Page Scheduling for Hybrid DRAM/NVM Memory Systems," in *14th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 12th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2023)*, 2023.

[51] ——, "Memory management methodology for application data structure refinement and placement on heterogeneous dram/nvm systems," in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022, pp. 748–753.

[52] ——, "Performance, Energy and NVM Lifetime-Aware Data Structure Refinement and Placement for Heterogeneous Memory Systems," Jul. 2023. [Online]. Available: https://www.techrxiv.org/articles/preprint/Performance_Energy_and_NVM_Lifetime-Aware_Data_Structure_Refinement_and_Placement_for_Heterogeneous_Memory_Systems/23628924

[53] P. Zardoshti *et al.*, "Understanding and improving persistent transactions on optane™ dc memory," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 348–357.