



## 저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

Doctoral Thesis

Leveraging Emerging Hardware to Improve the  
Performance of Data Analytics Frameworks

Moohyeon Nam

Department of Electrical and Computer Engineering  
Computer Science and Engineering

Graduate School of UNIST

2019

# Leveraging Emerging Hardware to Improve the Performance of Data Analytics Frameworks

Moohyeon Nam

Department of Electrical and Computer Engineering  
Computer Science and Engineering

Graduate School of UNIST

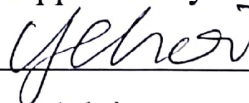
# Leveraging Emerging Hardware to Improve the Performance of Data Analytics Frameworks

A dissertation  
submitted to the Graduate School of UNIST  
in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy

Moohyeon Nam

12. 12. 2018

Approved by



Advisor

Young-ri Choi



# Leveraging Emerging Hardware to Improve the Performance of Data Analytics Frameworks

Moohyeon Nam

This certifies that the dissertation of Moohyeon Nam is approved.

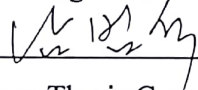
12. 12. 2018

signature



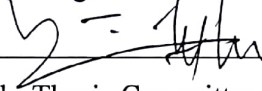
Advisor: Young-ri Choi

signature



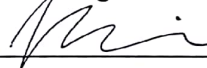
Beomseok Nam: Thesis Committee Member #1

signature




Sam H. Noh: Thesis Committee Member #2

signature



Woongki Baek: Thesis Committee Member #3

signature



Won-Ki Jeong: Thesis Committee Member #4

## Abstract

The data analytics frameworks have evolved along with the growing amount of data. There have been numerous efforts to improve the performance of the data analytics frameworks including MapReduce frameworks and NoSQL and NewSQL databases. These frameworks have various target workloads and their own characteristics; however, there is common ground as a data analytics framework. Emerging hardware such as graphics processing units and persistent memory is expected to open up new opportunities for such commonality. The goal of this dissertation is to leverage emerging hardware to improve the performance of the data analytics frameworks.

First, we design and implement *EclipseMR*, a novel MapReduce framework that efficiently leverages an extensive amount of memory space distributed among the machines in a cluster. EclipseMR consists of a decentralized DHT-based file system layer and an in-memory cache layer. The in-memory cache layer is designed to store both local and remote data while balancing the load between the servers with proposed *Locality-Aware Fair* (LAF) job scheduler. The design of EclipseMR is easily extensible with emerging hardware; it can adopt persistent memory as a primary storage layer or cache layer, or it can adopt GPU to improve the performance of map and reduce functions. Our evaluation shows that EclipseMR outperforms Hadoop and Spark for various applications.

Second, we propose  $B^3$ -tree and *Cache-Conscious Extendible Hashing* (CCEH) for the persistent memory. The fundamental challenge to design a data structure for the persistent memory is to guarantee consistent transition with 8-bytes of fine-grained atomic write with minimum cost.  $B^3$ -tree is a fully persistent hybrid indexing structure of binary tree and B+-tree that benefits from the strength of both in-memory index and block-based index, and CCEH is a variant of extendible hashing that introduces an intermediate layer between directory and buckets to fully benefit from a cache-sized bucket while minimizing the size of the directory. Both of the data structures show better performance than the corresponding state-of-the-art techniques.

Third, we develop a data parallel tree traversal algorithm, *Parallel Scan and Backtrack* (PSB), for k-nearest neighbor search problem on the GPU. Several studies have been proposed to improve the performance of the query by leveraging GPU as an accelerator; however, most of the works focus on the brute-force algorithms. In this work, we overcome the challenges of traversing multi-dimensional hierarchical indexing structure on the GPU such as tiny shared memory and runtime stack, irregular memory access pattern, and warp divergence problem. Our evaluation shows that our data parallel PSB algorithm outperforms both the brute-force algorithm and the traditional branch and bound algorithm.



## Contents

I	Introduction . . . . .	1
1.1	Thesis and Contributions . . . . .	3
1.2	Thesis Organization . . . . .	5
II	Background and Related Work . . . . .	6
2.1	EclipseMR: Distributed and Parallel Task Processing with Consistent Hashing . . . . .	6
2.2	$B^3$ -tree: Byte-Addressable Binary B-Tree for Persistent Memory . . . . .	7
2.3	Write-Optimized Dynamic Hashing for Persistent Memory . . . . .	8
2.4	Parallel Tree Traversal for Nearest Neighbor Query on the GPU . . . . .	11
III	EclipseMR: Distributed and Parallel Task Processing with Consistent Hashing . .	15
3.1	EclipseMR . . . . .	15
3.2	Distributed In-Memory Cache . . . . .	17
3.3	Evaluation . . . . .	19
IV	Index Structures for the Persistent Memory . . . . .	24
4.1	$B^3$ -Tree: Byte-addressable Binary B-tree . . . . .	24
4.2	Failure-Atomic $B^3$ -Tree Node Update . . . . .	25
4.3	Evaluation . . . . .	34
4.4	Cacheline-Conscious Extendible Hashing . . . . .	42
V	Parallel Tree Traversal for Nearest Neighbor Query on the GPU . . . . .	56
5.1	Parallel Scan and Backtrack for kNN Query . . . . .	56
5.2	Bottom-up Construction of SS-tree . . . . .	58
5.3	Experiments . . . . .	61
VI	Conclusion . . . . .	69
	References . . . . .	70
	Acknowledgements . . . . .	79

## List of Figures

1	<i>Extendible Hash Table Structure</i> . . . . .	9
2	<i>Data parallel tree traversal algorithms access a single tree node at a time and each thread determines whether each branch can be pruned out or not. But in task parallel tree traversal algorithms, each thread accesses different tree nodes and it causes significant warp divergence.</i> . . . . .	13
3	<i>Double-layered Chord ring in EclipseMR. The outer layer is the distributed in-memory cache layer and the inner layer is the distributed file system layer.</i> . . . .	16
4	<i>MapReduce Job Scheduling in EclipseMR</i> . . . . .	18
5	<i>IO throughput with varying the number of data nodes</i> . . . . .	20
6	<i>Performance comparison against Hadoop and Spark</i> . . . . .	21
7	<i>Execution Time of Iterative Jobs</i> . . . . .	22
8	<i>Page Structure of <math>B^3</math>-tree</i> . . . . .	24
9	<i>Insertion into <math>B^3</math>-tree Page</i> . . . . .	27
10	<i>Failure-Atomic Page Split in <math>B^3</math>-tree</i> . . . . .	29
11	<i>Failure-Atomic Redistribution</i> . . . . .	31
12	<i>Insertion Performance with Varying Page Sizes</i> . . . . .	35
13	<i>Search Performance: Balanced Trees vs Skewed Trees</i> . . . . .	35
14	<i>Deletion Performance with Varying Page Sizes</i> . . . . .	36
15	<i>Throughput Comparison with Varying Number of Indexed Data</i> . . . . .	37
16	<i>Insertion Performance Comparison (Latency)</i> . . . . .	38
17	<i>Search Performance with Varying Read Latency</i> . . . . .	39
18	<i>Insertion(a) and Search(b) Throughputs with Varying Number of Threads</i> . . . .	39
19	<i>Performance Comparison with 4 Threads(l) and 16 Threads(r) Using Mixed Workload</i> . . . . .	39
20	<i>Cacheline-Conscious Extendible Hashing</i> . . . . .	42
21	<i>Failure-Atomic Segment Split Example</i> . . . . .	43
22	<i>MSB segment index makes adjacent directory entries be modified together when a segment splits</i> . . . . .	46
23	<i>Buddy Tree Traversal for Recovery</i> . . . . .	47
24	<i>Throughput with Varying Segment/Bucket Size</i> . . . . .	50
25	<i>Breakdown of Time Spent for Insertion While Varying R/W latency of PM</i> . . .	52

26	<i>Performance of concurrent execution: latency CDF and insertion/search throughput</i> . . . . .	54
27	<i>Massively Parallel Scanning and Backtracking:</i> In the root node $A$ , the pruning distance is initially infinite. In step ①, we search a leaf node which is closest to the query point and update the pruning distance while computing the MAXDIST of child nodes. In the second tree traversal ②, regardless of whether $B$ or $C$ is closer to the query point, we fetch the leftmost child node $B$ from global memory if both $B$ and $C$ are within pruning distance. In node $B$ , we check which child nodes are within the pruning distance. In the example, suppose $D$ and $E$ are within the pruning distance. Then ③ we fetch the left child node $D$ . Suppose $H$ is not within the pruning distance. Then node $H$ will be pruned out, and ④ we visit node $I$ and kNN points will be updated. After processing node $I$ , ⑤ we fetch its sibling nodes $J$ and $K$ . If node $K$ does not update kNN points or pruning distance, ⑥ we fetch $K$ 's parent node $E$ from global memory and prune out child nodes ( $L$ in the example) which are farther than the pruning distance. If $M$ 's MINDIST is smaller than the pruning distance, ⑦ we fetch $M$ and ⑧ keep scanning its sibling nodes. If node $N$ does not update kNN points, ⑨ we move to its parent node $F$ , which does not have any child nodes within the pruning distance. Thus ⑩ we move one level up to node $C$ . If node $C$ does not have any child nodes within the pruning distance, we will move to the root node and finish the search. Otherwise, as in the example, we visit the leftmost leaf node $G$ as it is within the pruning distance. We keep this traversal and visit $G$ , $R$ , and $S$ . . .	57
28	<i>Bottom-up Constructed SS-trees vs Top-down Constructed SR-tree (Parent Link Tree Traversal)</i> . . . . .	61
29	<i>Distribution of Datasets Projected to the First Two Dimensions (<math>N</math>: number of clusters, <math>\sigma</math>: standard deviation)</i> . . . . .	62
30	<i>Query Processing Performance with Varying Input Distribution (100 clusters)</i> . .	63
31	<i>Query Processing Performance with Varying Number of Fan-outs</i> . . . . .	64
32	<i>Performance with Varying Dimensions (Synthetic Datasets (100 clusters))</i> . . . .	65
33	<i>Query Processing Performance with Varying <math>k</math></i> . . . . .	65
34	<i>Query Processing Performance with Real Datasets (NOAA)</i> . . . . .	66

# List of Tables

## I Introduction

The amount of data is continuously growing at this moment and demanding data analytics frameworks to be more efficient and scalable to handle such an enormous amount of data. There have been many studies on how to process the increasing data size, including MapReduce frameworks and NoSQL and NewSQL databases. The conventional data analytics heavily relies on the database systems that model the data into tables and their relations with others so that it can provide the structured query language (SQL) interface to manipulate the records and schema of the tables with atomicity, consistency, isolation, and durability. However, as the substantial portion of the generated data is unstructured and difficult to be molded into tabular form, the MapReduce programming model becomes very popular as it provides a new abstraction for programmers to handle the data in an easy and scalable way without the messy details that complicate the problem. In the MapReduce programming model, programmers can define their own custom map and reduce functions to process data in a scalable and flexible fashion. NoSQL databases are based on this MapReduce programming model to provide simplicity of design, simpler horizontal scaling, and finer control over availability with relaxed consistency model.

However, there are still needs for strong transactional properties and consistency requirements that the conventional database systems used to provide. NewSQL databases are modern relational databases that satisfy both the ACID guarantees of transactional database systems and the scalability of the NoSQL databases. These systems have different requirements to fulfill and their own unique workloads, but the frameworks have a common ground as data analytics frameworks. In this dissertation, we investigate how to leverage emerging hardware to improve the performance of the data analytics frameworks.

The frameworks have been built on the assumption of block-addressable storage devices, such as HDD or SSD for decades, which are slow, coarse-grained, but persistent. Despite the drawbacks of the devices, it is necessary to rely on the devices to ensure the durability of the data. Fast, byte-addressable DRAM has been used as a buffer to mitigate such deficiencies, but there is always a trade-off depending on which data is stored on which device due to the limited capacity of expensive memory space and volatile nature of the device. However, as byte-addressable persistent memory such as 3D Xpoint, phase-change memory, and STT-MRAM emerges, now the main memory is not only a volatile buffer but also persistent primary storage itself. The persistent memory is expected to provide low-latency as DRAM, but it persists data in the unit of bytes with large storage capacity.

As this byte-addressable persistent memory breaks the barrier separating volatile and non-volatile storage, it is necessary to design new data structures for this new hierarchy of memory. The persistent memory requires the data on the CPU cache to be written back to the memory to ensure the safety of the data, but unfortunately, the unit of atomic write from the cache to the memory is only 8-byte. Furthermore, as the order of the write instructions is not guaranteed, it is difficult to provide the consistency of the data structures. These challenges should be



dealt with the fine-grained memory management scheme with memory fence and cacheline flush instructions which barriers the memory write to keep the order between the memory operations and triggers the flush of cacheline containing the modified data, respectively.

There have been many studies to leverage the byte-addressability, durability, and high performance of persistent memory by re-designing block-based data structures such as B+-trees. B+-tree variants have been widely adopted in the past decades as they are designed to provide failure-atomicity, consistency, durability, and concurrency by being updated in block-granularity. These properties were not the primary concerns of the various byte-addressable in-memory data structures such as Radix-tree, Skip-list, and T-tree, since they were designed for the volatile main memory. These data structures are not cache-conscious as old processors have tiny CPU cache, but as CPU cache size has increased, in-memory B-tree variants that are conscious of cache locality and arrange sorted keys in hierarchical blocks have been developed [1, 2, 3]. The legacy in-memory indexes are outperformed by these variants of B-tree as they benefit from various features of modern processors such as large CPU caches, instruction-level parallelism, and memory-level parallelism, which means that block-based data structures are efficient as not only disk-based index, but also in-memory index.

However, B-tree variants are not always the correct solutions due to the growing size of the tree. There are latency-critical applications that can benefit from the byte-addressable persistent memory. The increasing level of the tree structure results in increased latency, which is unacceptable for such applications. Hash-based index structures have static flat structures that guarantee constant lookup time, which is appropriate for such latency-critical applications. Only a few of studies have attempted to adapt hash-based data structures to persistent memory. One of the main challenges in a hash-based structure for the persistent memory is in achieving efficient *dynamic rehashing* under the fine-grained failure-atomicity constraint. Dynamic rehashing is inevitable as predicting the capacity of a hash table is not always possible that the table might suffer from hash collisions, overflows, and under-utilization. However, rehashing is not desirable as it degrades total system throughput as the table is not accessible during the rehashing process, which significantly increases the tail latency of queries. Furthermore, rehashing requires a large number of writes to persistent memory, and the writes are expected to induce higher latency and energy consumption in persistent memory.

On the other hand, as the modern processor manufacturers focus on integrating more and more cores inside a single chip to increase the performance of a processor, the multi-core processors have become commodity hardware nowadays, and even the graphics processing units (GPUs) are widely adopted as SIMD (Single Instruction Multiple Data) accelerators to enhance the performance of the computations. It is inevitable to take into account the multiple levels of parallelism to design a data structure; however, it is a difficult task to efficiently exploit multiple cores with limited bandwidth and capacity of the memory hierarchy. Problems become even worse if we want to traverse tree-based indexes on the GPUs. One of the challenges in traversing a hierarchical index on the GPU is the tiny shared memory and runtime stack on the GPU. It

is necessary to store a tree node that is big enough to efficiently exploit the SIMD units of the GPU; however, the size of such tree node is too huge for the runtime stack to store multiple nodes to traverse the tree structure. The irregular memory access pattern of the tree traversal hinders exploiting parallelism since GPUs are designed for deterministic memory accesses. Traversing the hierarchical structure requires lots of branch operations that can cause the warp divergence problem that can decrease both the SIMD efficiency and the GPU utilization. To alleviate such problems, we have to take advantage of spatial locality and avoid cache invalidation by exploiting the byte-addressability and fine-grained memory management.

## 1.1 Thesis and Contributions

In this dissertation, I will show that *the data structures for the new emerging hardware should exploit fine-granularity to improve the performance of the data analytics frameworks while sustaining the consistency and increasing the concurrency*. To support this goal, I develop and evaluate a set of data structures and techniques to fully leverage the emerging hardware for the data analytics frameworks. The key contributions are summarized as follows:

- **EclipseMR: Distributed and Parallel Task Processing with Consistent Hashing**

We design and implement *EclipseMR*, a MapReduce framework that can efficiently utilize large distributed memory space in a cluster and benefit from emerging hardware with the following set of techniques. EclipseMR has two layers of consistent hash rings, which are a decentralized DHT-based file system and an in-memory key-value store with consistent hashing. The in-memory key-value store is designed to cache local data and remote data so that it can balance the load between the servers in a cluster. In order to efficiently leverage huge memory space distributed across the cluster with a higher cache hit ratio, we propose a *locality-aware fair* (LAF) job scheduler that acts as the load balancer for the cache. The LAF job scheduler predicts the availability of reusable data and assigns tasks to the servers with such data while balancing data locality and load balance. Our evaluation shows that EclipseMR is faster than Hadoop and Spark by a large margin for various applications.

- **$B^3$ -tree: Byte-Addressable Binary Block Tree for Persistent Memory**

The primary challenge in designing a B+-tree for fast, byte-addressable persistent memory is transforming a consistent state of the structure into another consistent state with 8-bytes of atomic write. Previous studies employ the append-only updates with additional metadata to manage the order of the keys or the selective persistence with hybrid memory hierarchy of DRAM and PM.

We design and implement the  $B^3$ -tree, a fully persistent hybrid indexing structure of binary tree and B+-tree that benefits from the failure-atomicity and byte-addressability of an in-memory index and the durability, cacheilne consciousness, and balanced tree height of a

block-based index. We also develop a logging-less failure-atomic split and merge algorithms for the  $B^3$ -tree that significantly reduce the number of cacheline flush instructions caused by the logging or journaling.  $B^3$ -tree consistently outperforms wB+-tree, one of the state-of-the-art B+-tree variant, by a large margin and shows comparable performance with partially persistent FPtree.

- **Write-Optimized Dynamic Hashing for Persistent Memory**

In the past few years, many studies have been proposed to leverage the byte-addressability, durability, and low latency of persistent memory, including, numerous variants of B+-tree indexes due to the fine-grained I/O for persistent memory. However, only a few have paid their attention to the hash-based index structures, which have multiple advantages including constant lookup time and higher memory utilization. The primary challenge of a hash-based index structure in persistent memory is to achieve efficient dynamic rehashing under fine-grained failure-atomicity constraint.

In this work, we adapt the extendible hashing for byte-addressable persistent memory by using cacheline-sized buckets and introducing an intermediate layer of segments to the extendible hashing. The three-level structure of this *Cache-Conscious Extendible Hashing* (CCEH) guarantees to find a record with only two cacheline accesses. We also develop a failure-atomic rehashing and recovery algorithm for CCEH without using explicit logging. We evaluate the performance with the state-of-the-art hashing techniques for the persistent memory, and our CCEH successfully reduces the maximum query latency by over two-thirds compared to the state-of-the-art hashing techniques.

- **Parallel Tree Traversal for Nearest Neighbor Query on the GPU**

The nearest neighbor search is a fundamental problem that finds the closest point to a given query point in multi-dimensional space, and it is used in wide application domains such as computer graphics, information retrieval, and scientific data processing. Recent advances of GPGPU (General-Purpose computing on Graphics Processing Units) computing, several studies have been proposed to accelerate the k-nearest neighbor search using GPUs, but most of the works focus on enhancing the exhaustive search for the exact k-nearest neighbors as it is known that the multi-dimensional hierarchical indexing trees are not suitable for the GPUs due to the tiny shared memory and runtime stack, irregular memory access pattern of tree traversal, and warp divergence problem.

In this work, we develop a data parallel tree traversal algorithm, *Parallel Scan and Backtrack* (PSB), for k-Nearest Neighbor query processing on the GPU. The PSB algorithm avoids the warp divergence problems while traversing the multi-dimensional tree-structured index to enhance SIMD efficiency. The sibling leaf nodes are linearly scanned to take advantage of accessing contiguous memory blocks and reducing unnecessary backtracking to the parent nodes. We also develop the parallel bottom-up construction algorithm for the

fast SS-tree construction by parallelizing Ritter’s minimum enclosing circle algorithm [4]. Our evaluation shows that the PSB algorithm outperforms not only the exhaustive brute-force search but also traditional branch and bound algorithms by a large margin.

## 1.2 Thesis Organization

The rest of this dissertation is organized as follows. In section II, we present the background and related work. In section III, we present a novel MapReduce framework EclipseMR that can benefit from the following emerging hardware. In section IV, we present index structures for the persistent memory that overcomes the challenge of failure-atomicity. In section V, we present the multi-dimensional indexing structure, SS-tree and its traversal algorithm for the GPU, called parallel scan and backtrack algorithm. In section VI, we conclude this dissertation.

## II Background and Related Work

### 2.1 EclipseMR: Distributed and Parallel Task Processing with Consistent Hashing

As the demand for large-scale data analysis frameworks grew in the high performance computing community in the late '90s, several distributed and parallel data analysis frameworks such as Active Data Repository [5], which supports the MapReduce programming paradigm and DataCutter [6], which supports generic DAG workflows, were developed for large scale scientific datasets. A few years later, industry had a growing demand for large-scale data processing applications and Google developed Google File System [7] and the MapReduce framework [8]. Since then, there has been a great amount of effort to extend and improve distributed job processing frameworks for various data-intensive applications. [9, 10, 11, 12, 13]

Spark [14, 13] shares the same goal as our framework in that it reuses a working set of data across multiple parallel operations. Resilient Distributed Datasets (RDDs) in Spark are read-only in-memory data objects that can be reused for subsequent MapReduce tasks. Spark addresses the conflict between job scheduling fairness with data locality by delaying a job for a small amount of time if the job can not launch a local task [15]. Our EclipseMR job scheduling is different from Spark in that EclipseMR employs consistent hashing to determine where to store and access the cached data objects. Based on the consistent hashing, EclipseMR strikes a balance between load balancing and data locality. *Dache* is another MapReduce framework where a central cache manager uses its best efforts to reuse the cached results of previous jobs [16]. Compared to *Dache*, EclipseMR is more scalable as it does not have a central directory to keep the list of cached data objects that can change dynamically at a very fast pace.

*Main Memory MapReduce (M3R)* proposed by Shinnar et al. [12] is a MapReduce framework that performs in-memory shuffle by simply storing the intermediate results of map tasks in main memory instead of the block device storage. They show in-memory shuffle significantly improves a certain type of applications. however, M3R can not be used if workloads are large and do not fit in main memory or applications require *resilience* because the in-memory framework is not fault tolerant. Moreover, it is questionable if MapReduce is the right programming paradigm for their target application - sparse matrix vector multiplication. Rahman et al. [17] proposed *HOMR* - a hybrid approach to achieve the maximum possible overlapping across map, shuffle, and reduce phases. Our work is similar to that in the sense that EclipseMR aggressively overlap three phases by proactive shuffling. The in-memory caching layer of EclipseMR is similar to *Tachyon*, which is the cache layer that sits on top of HDFS and acts as a distributed cache for Spark. The difference between *Tachyon* and EclipseMR in-memory caching is that EclipseMR caching evenly distributes popular cached objects via LAF algorithm.

DryadInc [10] is an incremental computation framework that allows computations to reuse

partial results of the computations from previous runs. Tiwari et al. [18] proposed the *MapReuse* delta engine as an in-memory MapReduce framework that detects input data similarity, reuses available cached intermediate results and computes only for the new portion of input data. They show the reuse of intermediate results significantly improves job execution time. *ReStore* [19] is another framework that stores intermediate results generated by map tasks in HDFS so that they can be reused by subsequent jobs. Their works are similar to ours but the conventional fair job scheduling policies they use do not guarantee balancing the workloads if requested intermediate results are available only on a small number of overloaded servers.

The *MRShare* framework proposed by Nykiel et al. [20] merges a batch of MapReduce queries into a single query so that it takes the benefits of sharing input and output data across multiple queries. The multiple query optimization problem has been extensively studied in the past, and it has been proven to be an NP problem. Nevertheless, extensive research has been conducted to minimize query processing time through data and computation reuse using heuristics or probabilistic efforts [19, 21, 10, 22]. These multiple query optimization studies are complementary to our work.

## 2.2 $B^3$ -tree: Byte-Addressable Binary B-Tree for Persistent Memory

As byte-addressable persistent memory is now on the horizon, numerous studies have been conducted to exploit new opportunities of its beneficial features in various domains including file systems and database management systems [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38].

Systems on byte-addressable persistent memory has to safeguard against failures with fine-grained write atomicity. That is, in persistent memory, the granularity of failure-atomic writes is expected to be 8-bytes, or a cache line if we use hardware transactional memory [39, 33]. In persistent memory, Such a small write granularity makes it difficult to guarantee consistency of various data structures including B+-trees because not all 8-byte writes can transform a consistent index into another consistent index. Moreover, independent store instructions can be arbitrarily reordered in modern processors. Therefore, a large number of expensive `clflush` and `mfence` instructions are required to guarantee the consistency of data structures [35].

To resolve this problem, Venkataraman et al. [35] proposed to use multi-versioning scheme so that we can roll back to previous consistent states. Version-based recovery methods, though, have a limitation in that it requires an expensive garbage collection.

Alternatively, Yang et al. [37] proposed NV-Tree that updates B-tree pages in *append-only* manner instead of making a large portion of data structures dirty so that it can reduce the number of calls to `clflush` and `mfence`. While the append-only update mitigates the overhead of write transactions, read transactions suffer from finding a key in unsorted arrays. The append-only update scheme has been adopted by other persistent B-tree variants developed later, i.e., wB+-tree [40] and FP-tree [41]. wB+-tree proposed to use additional metadata to manage

the ordering of keys in order to resolve the problem of unsorted keys. However, the additional metadata in wB+-tree requires additional cache line flushes, which is sub-optimal. We note that wB+-tree still relies on expensive logging methods for tree rebalancing operations because it stores both internal and leaf tree pages in persistent memory.

FPTree, proposed by Oukid et al., is similar to NV-Tree in that it also stores leaf pages in persistent memory while keeping internal tree pages in DRAM [41]. Different from NV-Tree, FPTree exploits hardware transactional memory to efficiently handle concurrency of internal tree page accesses and it reduces the cache miss ratio via fingerprinting, which is one-byte hash value for keys stored in leaf page. Both NV-Tree and FPTree employ *selective persistence* where they store internal pages in volatile DRAM but leaf pages in persistent memory. Therefore, the internal pages in NV-Tree and FP-Tree may be lost upon system failure. Although the internal pages can be reconstructed from scratch using the leaf pages in persistent memory, the entire reconstruction process can hinder the instant use of the index.

Every 8-byte store instruction used in the FAST and FAIR algorithms transforms a B+-tree into another consistent state or a transient inconsistent state that read operations can tolerate. By making read operations tolerate transient inconsistency, FAST and FAIR B-tree avoids expensive copy-on-write and logging. However, a large number of shift operations employed in FAST and FAIR B-tree make the performance degrade as we increase the tree node size and aggravate wearing issues of persistent memory.

### 2.3 Write-Optimized Dynamic Hashing for Persistent Memory

The focus of this subsection is on dynamic hashing, that is, hashing that allows the structure to grow and shrink according to need. While various methods have been proposed [42, 43, 44], our discussion concentrates on extendible hashing as this has been adopted in numerous real systems [45, 46, 47, 48, 49] and as our study extends it for PM.

**Extendible Hashing:** Extendible hashing was developed for time-sensitive applications that need to be less affected by full-table rehashing [50]. In extendible hashing, re-hashing is an incremental operation, i.e., rehashing takes place per bucket as hash collisions make a bucket overflow. Since extendible hashing allocates a bucket as needed, pointers to dynamically allocated buckets need to be managed in a hierarchical manner as in B-trees in such a way that the split history can be kept track of. This is necessary in order to identify the correct bucket for a given hash key.

Figure 1 shows the legacy design of extendible hashing. In extendible hashing, a hash bucket is pointed to by an entry of a *directory*. The directory, which is simply a *bucket address table*, is indexed by either the leading (most significant) or the trailing (least significant) bits of the key. In the example shown in Figure 1, we assume the trailing bits are used as in common practice and each bucket can store a maximum of five key-value records. The *global depth*  $G$  stores the number of bits used to determine a directory entry. Hence, it determines the *maximum* number of buckets, that is, there are  $2^G$  directory entries. When more hash buckets are needed,



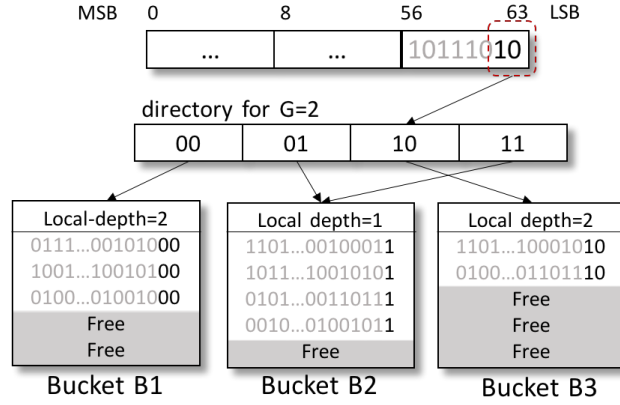


Figure 1: *Extendible Hash Table Structure*

extendible hashing doubles the size of the directory by incrementing  $G$ . From the example,  $G$  is 2, so we use the low end 2 bits of the key to designate the directory entry in the directory of size 4 ( $2^2$ ). Eventually, when the buckets fill up and split, needing more directory entries,  $G$  can be incremented to 3, resulting in a directory of size 8.

While every directory entry points to a bucket, a single bucket may be pointed to by multiple directory entries. Thus, each bucket is associated with a *local depth* ( $L$ ), which indicates the length of the common hash key in the bucket. If a hash bucket is pointed by  $k$  directory entries, the local depth of the bucket is  $L = G - \log_2 k$ . For example in Figure 1, B2 is being pointed to by 2 directory entries. For this bucket, as the global depth ( $G$ ) is 2 and the bucket is pointed to by two directory entries, the local depth of the bucket ( $L$ ) is 1.

When a hash bucket overflows, extendible hashing compares its local depth against the global depth. If the local depth is smaller, this means that there are multiple directory entries pointing to the bucket, as for bucket B2 in Figure 1. Thus, if B2 overflows, it can be split without increasing the size of the directory by dividing the directory entries to point to two split buckets. Thus,  $G$  will remain the same, but the  $L$ s for the two resulting buckets will both be incremented to 2. In the case where the bucket whose local depth is equal to the global depth overflows, i.e., B1 or B3 in Figure 1, the directory needs to be doubled. In so doing, both the global depth and the local depth of the two buckets that result from splitting the overflowing bucket also need to be incremented. Note, however, that in so doing, overhead is small as rehashing of the keys or moving of data only occur for keys within the bucket. With the larger global and local depths, the only change is that now, one more bit of the hash key is used to address the new buckets.

The main advantage of extendible hashing compared to other hashing schemes is that the rehashing overhead is independent of the index size. Also, unlike other static hash tables, no extra buckets need to be reserved for future growth that results in extendible hashing having higher space utilization than other hashing schemes [51]. The disadvantage of extendible hashing is that each hash table reference requires an extra access to the directory. Other static hashing schemes do not have this extra level of indirection, at the cost of full-table rehashing. However,



it is known that the directory access incurs only minor performance overhead [52, 51].

**PM-based Hashing:** Recently a few hashing schemes, such as *Level Hashing* [53], *Path Hashing* [54], and PCM-friendly hash table (PFHT) [55] have been proposed for persistent memory as the legacy in-memory hashing schemes fail to work on persistent memory due to the lack of consistency guarantees. Furthermore, persistent memory is expected to have limited endurance and asymmetric read-write latencies. We now review these previous studies.

PFHT is a variant of bucketized cuckoo hashing designed to reduce write accesses to PCM as it allows only one cuckoo displacement to avoid cascading writes. The insertion performance of cuckoo hashing is known to be about 20~ 30% slower than the simplest linear probing [56]. Furthermore, in cuckoo hashing, if the load factor is above 50%, the expected insertion time is no longer constant. To improve the insertion performance of cuckoo hashing, PFHT uses a stash to defer full-table rehashing and improve the load factor. However, the stash is not a cache friendly structure as it linearly searches a long overflow chain when failing to find a key in a bucket. As a result, PFHT fails to guarantee the constant lookup cost, i.e., its lookup cost is not  $O(1)$  but  $O(S)$  where  $S$  is the stash size.

Path hashing is similar to PFHT in that it uses a stash although the stash is organized as an inverted binary tree structure. With the binary tree structure, path hashing reduces the lookup cost. However, its lookup time is still not constant but in log scale, i.e.,  $O(\log B)$ , when  $B$  is the number of buckets.

Level hashing consists of two level hash tables. The top level and bottom level hash tables take turns playing the role of the stash. When the bottom level overflows, the records stored in the bottom level are rehashed to a  $4\times$  times larger hash table and the new hash table becomes the new top level, while the previous top level hash table becomes the new bottom level stash. Unlike path hashing and PFHT, level hashing guarantees constant lookup time.

While, level hashing is an improvement over previous work, our analysis shows that the rehashing overhead is no smaller than legacy static hashing schemes contrary to the authors' claim that it is reduced by  $1/3$ . As, at least one of the two hash tables is always almost full in level hashing, the bottom level hash table often fails to accommodate a collided record resulting in another rehash. The end result is that level hashing is simply performing a full-table rehash in two separate steps.

Consider the following scenario. Say, we have a top level hash table that holds 100 records and a bottom level stash holds 50 records. Hence, we can insert 150 records without rehashing if a hash collision does not occur. When the next 151th insertion incurs a hash collision in the bottom level, the 50 records in the bottom level stash will be rehashed to a new top level hash table of size 200 such that we have 150 free slots. After the rehash, subsequent 150 insertions will make the top level hash table overflow. However, since the bottom level hash table does not have free space either, the 100 records in the bottom level hash table have to be rehashed. To expand a hash table size to hold 600 records, level hashing rehashes total 150 records, that is, 50 records for the first rehashing and another 100 records for the second rehashing.

On the other hand, suppose the same workload is processed by a legacy hash table that can store 150 records as the initial level hash table does. Since 151th insertion requires more space in the hash table, we increase the hash table size by four times instead of two as the level hashing does for the bottom level stash. Since the table now has 600 free spaces, we do not need to perform rehashing until the 601th insertion. Up to this point, we performed rehashing only once and only 150 records have been rehashed.

Interestingly, the numbers of rehashed records are no different. We note that the rehashing overhead is determined by the hash table size, not by the number of levels. As we will show in Section 4.4.7, the overhead of rehashing in level hashing is no smaller than other legacy static hashing schemes.

## 2.4 Parallel Tree Traversal for Nearest Neighbor Query on the GPU

### 2.4.1 Stackless Tree Traversal

In computer graphics, a very large number of rays are concurrently traced by leveraging many GPU cores. In order for classic recursive tree traversal algorithms to traverse bounding volume hierarchies, the size of the run-time stack space must be as large as the maximum stack depth times the number of rays. However, the size of shared memory in modern GPUs is very small (less than 64KB). Therefore, the computer graphics community has proposed various stackless tree traversal algorithms such as *kd-restart* [57], *skip pointer* [58], rope tree [59], and short stack [60].

The Kd-restart algorithm proposed by Foley et al. [57] divides a query line into multiple small line segments while navigating a kd-tree. Thereby, it reduces the size of each bounding box of a line segment. Then, it repeatedly searches the kd-tree with the small bounding box from its root node again. This restart strategy eliminates backtracking and the need for a large run-time stack.

The rope tree [59] and the parent link [61] algorithms eliminate the need for a run-time stack by using auxiliary links. In rope tree, each node stores *ropes* - pointers to neighboring tree nodes in each dimension, thus a rope can be followed if a query line segment intersects a face of the bounding box. Unfortunately kd-restart and rope tree algorithms cannot be directly employed for kNN query processing because the kNN query is not a line segment. kNN query processing irregularly traverses an indexing tree and prunes out sub-trees based on the distance between a query point and the bounding volumes of sub-trees.

The *parent link* [61] algorithm is a more generic stackless tree traversal algorithm that can be employed for kNN query processing. In the parent link algorithm, each tree node has a pointer to its parent node (parent link) [61]. Instead of relying on the run-time stack, the parent link algorithm allows backtracking to a parent node by following the parent link pointer. A drawback of the parent link tree algorithm is that the same tree node has to be fetched from slow global memory multiple times when it backtracks.

*Skip pointer* is another stackless tree traversal algorithm that employs auxiliary pointers to a

right sibling node or a right sibling of its parent node. Unlike parent link, skip pointer does not allow backtracking to parent nodes. Instead, skip pointer visits right sibling nodes of the same parent. Only if it visits the rightmost sibling node of the current parent node, it backtracks to a higher level of the tree and visits the sibling of its parent node. The skip pointer algorithm is guaranteed to not visit previously accessed tree nodes, and it avoids fetching the same tree node multiple times from global memory. However, this is also a drawback of skip pointer algorithm because visiting all sibling nodes requires too many accesses to unnecessary tree nodes, especially for kNN query processing.

Another solution to the tiny run-time stack problem is to use a fixed size of small shared memory as a *short stack* [60]. If the short stack is not deep enough for a tree traversal, the short stack deletes a tree node from the bottom of the stack and pushes a new tree node on the top. While traversing a tree with a short stack, we may find a parent node has been evicted from the short stack. If the deleted parent node has to be visited for backtracking, the short stack algorithm restarts the tree traversal from the root node again, as in *kd-restart*. Although the short stack algorithm increases the chance of reusing previously visited tree nodes, it often restarts the tree traversal, which adds the overhead of fetching tree nodes from global memory. The overhead of global memory access is known to offset the benefits of reusing tree nodes available in the short stack [62]. Moreover, the size of a single SS-tree node becomes larger than 32 Kbytes when the dimension is higher than 32. Considering the tiny shared memory size of the GPU, the short stack algorithm cannot be used for high dimensional SS-trees.

#### 2.4.2 Data Parallelism vs Task Parallelism

The stackless tree traversal algorithms described in section 2.4.1, *kd-restart* [57], *skip pointer* [58], rope tree [59], and short stack [60] focus on distributing a large number of line intersection queries across a set of GPU processing units. Such *task parallelism* is known to improve query processing throughput, but it does not improve the query response time of individual queries. *Data parallelism* contrasts to task parallelism in the sense that a large number of GPU processing units perform the same task on different parts of an index.

Figure 2 illustrates how a data parallel algorithm and a task parallel algorithm traverse a tree structure in different ways. In data parallel tree traversal algorithms, a set of threads in a GPU block concurrently access the same tree node and cooperate to determine which child node to fetch and visit. But in a task parallel tree traversal algorithm, each thread processes a different query and follows a different search path. Therefore each GPU processing unit accesses different parts of global memory and the number of accessed tree nodes varies across GPU processing units.

In task parallelism, the query response times are determined by the slowest thread in the block because a block of threads has to wait until all the other threads in the same block finish. Moreover, task parallelism makes each thread execute a different instruction. But, a warp is the minimum thread scheduling unit in CUDA architecture. All threads in a warp must execute the

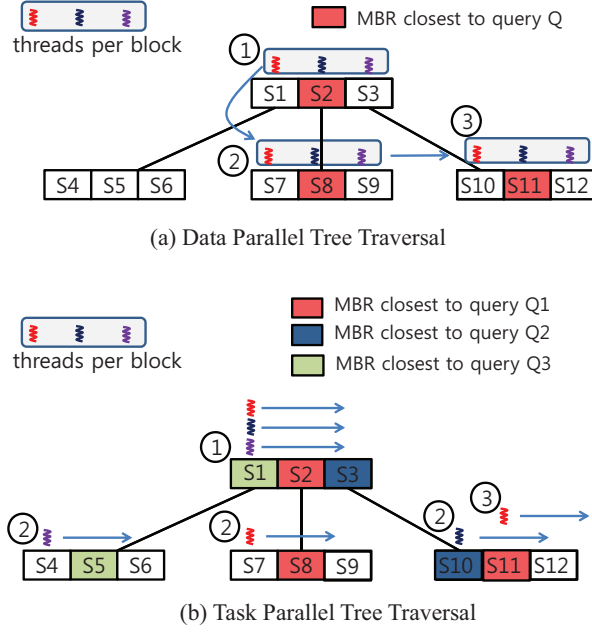


Figure 2: *Data parallel tree traversal algorithms access a single tree node at a time and each thread determines whether each branch can be pruned out or not. But in task parallel tree traversal algorithms, each thread accesses different tree nodes and it causes significant warp divergence.*

same instruction. If a thread in a warp needs to execute different instructions from the other threads, it needs to wait for multiple cycles until other threads finish. As more threads diverge, *SIMD efficiency* decreases and the utilization of GPU processing units decreases. This problem is called *warp divergence*.

In order to avoid such warp divergence and improve individual query response time, we develop a data parallel tree traversal algorithm and efficiently utilize a large number of GPU processing units.

### 2.4.3 SS-Tree

SS-tree [63] is a balanced  $n$ -ary multi-dimensional tree structure designed for nearest neighbor query processing. SS-tree employs bounding spheres instead of bounding rectangles for the shapes of tree nodes. Employing bounding spheres is not only beneficial for reducing the size of a tree node but it can also eliminate a number of conditional branches that are required by the classic *branch-and-bound* search algorithm [64].

SS-tree has been shown to outperform R-tree and K-D-B-tree for high dimensional datasets in many prior studies [63, 65]. Although bounding sphere volumes of SS-tree are often much larger than bounding rectangle volumes of R-trees especially in low dimensions, it is known that the number of visited tree nodes is often much smaller than that of R-trees in high dimensions [65]. Moreover, SS-trees can prune out child nodes with fewer computations than other indexing

structures. That is, rectangular bounding boxes in variants of R-tree or K-D-B-tree require the calculation of distances to each facet of a bounding shape. As the dimension increases, the number of facets also increases and the computation overhead to calculate the high-dimensional distances increases exponentially. Instead, SS-tree just computes the distance between a query and a centroid and adds or subtracts the radius of the bounding sphere, which significantly reduces the computation time.

The *incremental* kNN search algorithm [66] that uses a *priority queue* is known to perform faster than the branch-and-bound algorithm. However on the GPU, a block of threads share the priority queue, which necessitates protecting the priority queue using a lock. The lock will serialize a large number of threads, which results in high warp divergence and significant performance degradation.

### III EclipseMR: Distributed and Parallel Task Processing with Consistent Hashing

As the more space is available in the main memory, developers have started to adopt in-memory caching. Hadoop, which is one of the most popular MapReduce frameworks, also introduces in-memory caching in HDFS to cache local input data. However, the input data caching alone does not significantly improve the job execution time, especially for compute-intensive applications. In the database systems field, a considerable amount of studies has been conducted to show semantic caching successfully reduces query response time and improves system throughput by exploiting sub-expression commonality across multiple queries. If multiple MapReduce jobs have some common sub-computations, caching not only the input data but also the intermediate results for the sub-computations can greatly speed up subsequent tasks.

#### 3.1 EclipseMR

EclipseMR consists of a job scheduler, a resource manager, and double-layered consistent hash ring structures - a DHT file system and a distributed in-memory key-value store as shown in Figure 3. The job scheduler is responsible for assigning incoming queries, including MapReduce tasks, to back-end worker servers, and the resource manager is responsible for server join, leave, failure recovery, and file upload. The distributed in-memory cache and the DHT file system are completely decentralized components leveraging consistent hash rings. EclipseMR requires the job scheduler and resource manager to act as coordinators, but any worker server can take on the responsibility regardless. Hence, the job scheduler and the resource manager are selected by a distributed election algorithm.

##### 3.1.1 DHT File System

As in HDFS, DHT file system in EclipseMR partitions an input data file into fixed-sized blocks, but the partitioned data blocks are distributed across the servers according to their hash keys. Since the location of the partitioned blocks can be determined by hash functions, the DHT file system does not need a centralized directory service that manages the location of each block. Instead, we store metadata about a file including file name, owner, file size, and partitioning information in a decentralized manner. For example, if a user uploads a file, we generate a hash key using the file name, and store the metadata about the file in the server (*file metadata owner*) whose hash key range includes the file's hash key. At the same time, the partitioned file blocks are distributed across servers based on their hash keys.

Later, when an application wants to access a file, it obtains the hash key of the file using its file name, and accesses the file metadata owner in order to check the access permission, file size, hash keys of the partitioned blocks, etc. Once the applications reads the file metadata, it multicasts the block read requests to remote servers. Suppose the hash key ranges of the

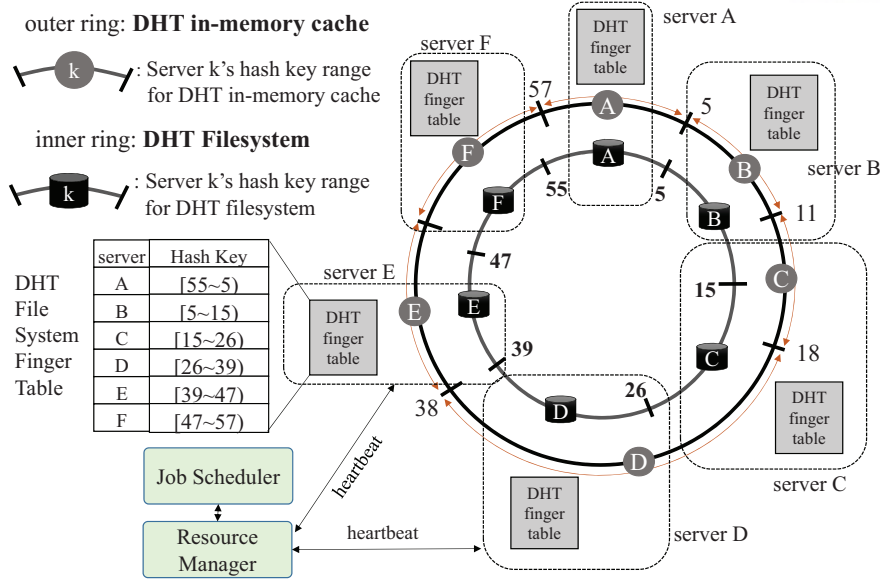


Figure 3: *Double-layered Chord ring in EclipseMR. The outer layer is the distributed in-memory cache layer and the inner layer is the distributed file system layer.*

DHT file system are as shown in Figure 3. If a file's hash key is in the range of  $[5, 15)$ , its file metadata will be stored in server  $B$ . To resolve the input data block skew problem, we distribute the partitioned file blocks across the ring using their hash keys.

We make DHT file system fault tolerant by replicating the file metadata as well as file blocks in predecessors and successors. When a worker server fails, either a predecessor or a successor will take over the faulty server and utilize the replicated blocks and metadata. Hence, unless a server fails along with its predecessor and successor at the same time, the DHT file system can tolerate system failures. If a resource manager or a scheduler fails, the rest of the worker servers execute an election algorithm to choose a new resource manager and a scheduler.

In the DHT file system, each server manages its own routing table, called finger table, containing  $m$  peer servers' information.  $m$  can be determined by system administrators but it should be chosen so that  $2^m - 1 > S$ , where  $S$  is the number of servers in the hash ring. Unless a cluster has more than thousands of servers, as in large scale peer-to-peer file sharing systems, we set  $m$  to the total number of servers to enable the *one hop DHT* routing [67]. When  $m$  is smaller, file IO requests can be redirected and the IO performance can be degraded. Because most distributed query processing systems are more stationary than dynamic peer-to-peer file sharing systems and the number of servers is usually less than a couple thousand, storing complete routing information for entire servers in a DHT routing table does not hurt the scalability of the system but improves data access performance [67].

Since the local DHT routing table is very small, the table lookup places minimal overhead on each server. When a server receives a file block access request from a remote server, it checks if the hash key of the file block is within its own hash key range. If so, it looks up its local



disks and serves the data access request. Otherwise, i.e., if zero hop routing is not enabled, it routes the request to another server that owns the hash key as in the classic DHT routing algorithm [68].

The DHT routing table is stationary so that it updates neighbor information including successor and predecessor only when a participating server joins, leaves, or fails. Each server exchanges heartbeat messages with direct neighbors to detect server failures, and the resource manager and job scheduler are notified when a server failure is detected. If a server fails, the resource manager reconstructs the lost file blocks in a *take-over* server using the replicated data blocks.

### 3.2 Distributed In-Memory Cache

In data-intensive computing, it is common for same applications to submit jobs that share the same input data. For example, database queries often access the same tables. There exist several prior works [69, 70] that report more than 30% of MapReduce jobs are repeatedly submitted in a production environment. Over the past decades, there have been a large number of works that exploit sub-expression commonality across multiple queries and incremental computation [19, 21, 20, 10, 22, 18]. The incremental computation significantly increases the chances of data reuse, reduces the job response time, and improves the system throughput.

On top of the DHT file system, EclipseMR deploys a distributed in-memory cache layer to exploit the incremental computation. The distributed in-memory cache consists of two partitions - *iCache* and *oCache*.

*iCache* is where input data blocks are implicitly cached. The latest HDFS also implemented in-memory caching, but HDFS in-memory caching stores only local input data blocks. Since data skew problem occurs not only in a record level but also in an input block level, HDFS in-memory caching does not mitigate the skew problem of input blocks. To resolve this problem, we let *iCache* allow input data blocks to be cached in peer servers according to their hash keys.

*oCache* is where intermediate results of map tasks and outputs of iterative jobs are explicitly cached by user applications. EclipseMR tags the cached data with their metadata (application ID, user-assigned ID for cached data). *oCache* helps avoid redundant computations by sharing the intermediate results among multiple jobs. *oCache* is similar to RDDS in Spark, but intermediate results or outputs for iterative jobs in EclipseMR are cached according to their hash keys, so it evenly distributes frequently accessed cached data objects across the entire distributed memories. The cached intermediate results and outputs of iterative jobs are also persistently stored in the DHT file system according to their hash keys so that long running jobs can survive faults and restart from the point of failure.

The hash key ranges of in-memory caches are determined by a job scheduler based on workload pattern so that popular hash key ranges can use more distributed memories. However, the hash key ranges of DHT file system are statically determined by consistent hashing and do not change unless servers join or fail. Therefore, the hash key ranges of the distributed in-memory



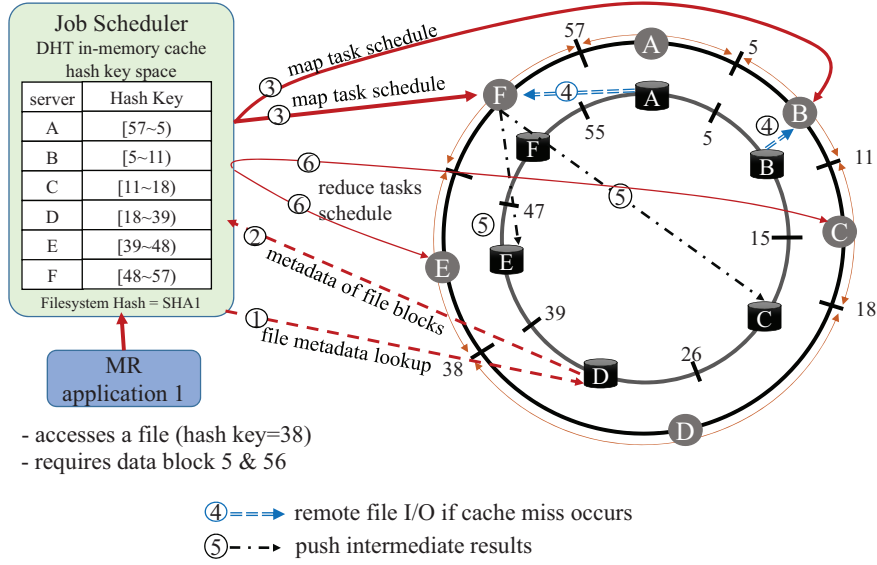


Figure 4: MapReduce Job Scheduling in EclipseMR

cache layer can be misaligned with the hash key ranges of the DHT file system.

### 3.2.1 MapReduce Processing

Figure 4 illustrates how EclipseMR processes a MapReduce job on a double layered ring structure. Suppose a job is submitted to the job scheduler. The job scheduler runs a hash function with the input file name to find out which server is the file metadata owner. Suppose server *D* is the file metadata owner in the example (①). Then, server *D* checks the file access permission and replies how the file is partitioned (step ②) and what are their hash keys.

If the input file is partitioned into two blocks and their hash keys are 6 and 56, the two blocks are stored in server *A* and *B*'s local disks, respectively. Given the hash keys of the two blocks, the job scheduler searches the hash key ranges of the distributed in-memory caches, and assigns a map task to each of server *B* and server *F* (step ③). Note that a map task is scheduled in server *F* instead of *A* even though an input file block 56 is stored in server *A*'s local disk. This is because each worker server's hash key range in the job scheduler's hash key table are misaligned with the hash key ranges of the DHT file system.

If server *F* has the input file block in its *iCache*, it reuses it. Otherwise, server *F* reads the input file block from the DHT file system, i.e., looks up its DHT routing table to find out the file block 56 exists in remote server *A*'s local disk. After reading the block from *A*'s local disks, server *F* stores the block in its *iCache* (step ④), and runs map tasks.

While map tasks are running, EclipseMR forwards the intermediate results generated by the map tasks to other servers according to the hash keys of the intermediate results so that they are persistently stored in the DHT file system (step ⑤). EclipseMR stores the intermediate results in persistent file systems as in Hadoop so that it can restart failed tasks and reuse the

intermediate results of the previous failed tasks. Although we store the intermediate results on disk, they can be cached in *oCache* for future reuse. Note that we store the intermediate results on the reducer side, not on the mapper side. The stored intermediate results are invalidated by time-to-live (TTL) which can be set by applications, and they are not replicated by default.

While map tasks are generating intermediate results, they notify the scheduler with their hash keys. With the given hash keys, the scheduler schedules reduce tasks where the intermediate results are stored. Reduce tasks read these intermediate results from *oCache* or the DHT file system using the hash keys (step ⑥).

If a user application specifies it can reuse intermediate results and they are available in *oCache* or the DHT file system, the map tasks skip computation and reducer tasks can immediately reuse the cached data. If intermediate results are not available, the map tasks search *iCache* for input data blocks to reuse. If input data blocks are not available either, they read input data blocks from the DHT file system. There exist certain applications such as **k-means** that can not reuse intermediate results between map tasks and reduce tasks, but they need the results of reduce tasks from each iteration. For such applications, the EclipseMR allows applications to store the iteration outputs in *oCache* or the DHT file system instead of intermediate results.

### 3.2.2 Proactive Shuffling

Hadoop stores the intermediate results in the local disks of the server where the map tasks run. The shuffle phase in Hadoop sorts, splits, and sends the intermediate results to reducers. It is known that the shuffle phase of MapReduce is network intensive and the shuffle phase can constitute a bottleneck. Hadoop tries to pipeline map, shuffle, and reduce phases by starting reduce tasks as soon as intermediate result files are available, but Hadoop pipelining is far from satisfactory, and there have been several previous works that try to aggressively overlap the shuffle phase with the map phase and decouple from the reduce phase [71, 72, 9, 73, 74].

Unlike Hadoop or Spark, EclipseMR determines where to run reduce tasks based on the hash keys of the intermediate results. Therefore, the shuffle phase in EclipseMR does not have to wait until map tasks finish. Instead, EclipseMR lets each mapper pipeline the intermediate results to the DHT file system in a decentralized fashion while they are being generated. Based on the hash keys of the intermediate results, each map task stores the intermediate results in a memory buffer for each hash key range. When the size of this buffer reaches a certain threshold specified by the application, EclipseMR spills the buffered results to the DHT file system so that they can be accessed by reducers.

## 3.3 Evaluation

In this section, we first evaluate the performance of EclipseMR by quantifying the impact of each feature we presented. We then compare the performance of EclipseMR against Hadoop 2.5 and Spark 1.2.0. We have implemented the core modules of EclipseMR prototype in about

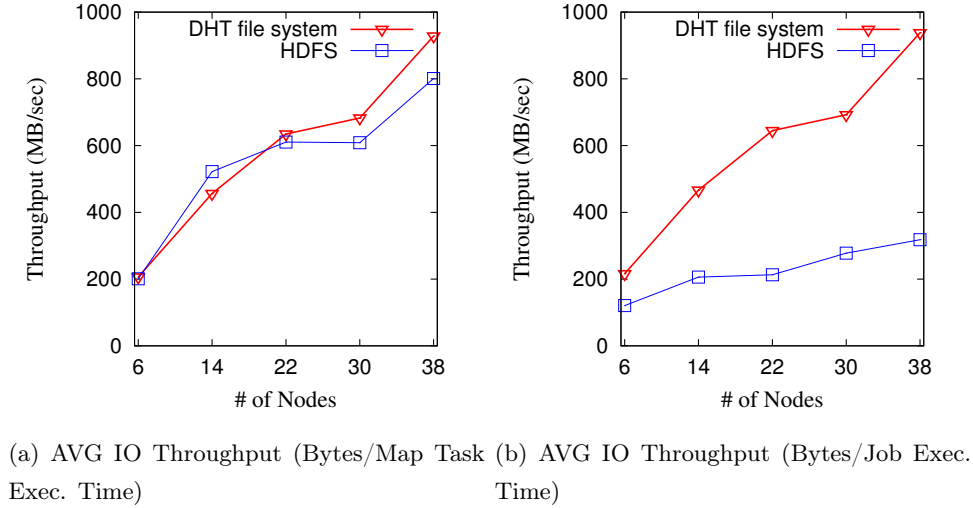


Figure 5: IO throughput with varying the number of data nodes

17,000 lines of C++ code. The source code that we used for the experiments is available at <http://github.com/DICL/EclipseMR>.

We run the experiments on a 40-nodes (640 vCPUs) Linux cluster that runs CentOS 5.5. Each node has dual Intel Xeon Quad-core E5506 processors, 20 GBytes DDR3 ECC memory, and a 5400rpm 256GB HDD for OS and a single 7200rpm 2TB HDD for HDFS and the DHT file system. 20 nodes are connected via a 1G Ethernet switch, the other 20 nodes are connected via another 1G Ethernet switch, and another 1G Ethernet switch forms the two level network hierarchy. We set both the number of map task slots and the number of reduce task slots to 8 (total 640 slots).

We use HiBench [75] to generate 250 GB text input datasets for the **word count**, **inverted index**, **grep**, and **sort** applications, 15 GB graph input datasets for **page rank**, and 250 GB **kmeans** datasets. In [76, 77], they report the median input sizes for the majority of data analytics jobs in Microsoft and Yahoo datacenters are under 14 GBytes. Hence, we also evaluate the performance of EclipseMR with small 15 GB text input datasets that we collect from Wikipedia and 15 GB k-means datasets that we synthetically generate with varying distributions.

### 3.3.1 IO Throughput

In the experiments shown in Figure 5(a), we measure the read throughput (total bytes/map task execution time) of the DHT file system and HDFS using HDFS DFSIO benchmark while varying the number of servers. As can be seen in Figure 5(a), HDFS and DHT file system show similar IO throughput. Note that this metric does not include the overhead of NameNode directory lookup and job scheduling, but it measures the read latency of local disks. In Figure 5(b), we measure the read throughput in a different way, i.e., total bytes/job execution time. While the DHT file system has negligible overhead in decentralized directory lookup and job scheduling, Hadoop suffers from various overheads including NameNode lookup, container initialization, and

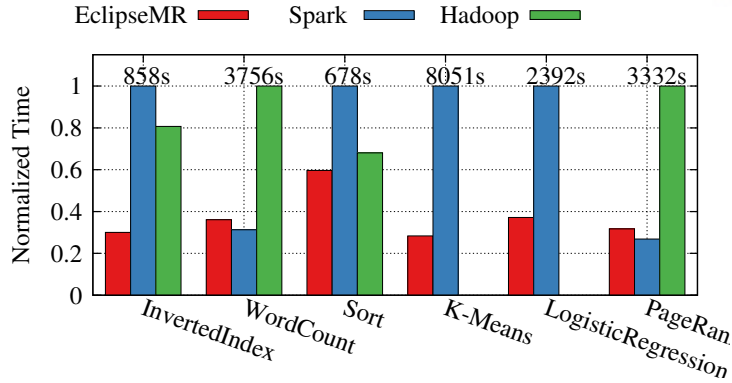


Figure 6: Performance comparison against Hadoop and Spark

job scheduling.

In order to evaluate the scalability of the DHT routing table and HDFS name node, we submitted multiple concurrent DFSIO jobs in an experiment that we do not show due to the page limit, and we observed that the IO throughput of HDFS degrades at a much faster rate than the DHT file system.

### 3.3.2 Comparison with Hadoop and Spark

Finally we compare the performance of EclipseMR against Hadoop and Spark. Since all three frameworks provide different *levers* for tuning, we performed Hadoop and Spark tunings to our best efforts according to various performance tuning guides available on the web.

However, it is hard to quantify which design of EclipseMR contributes to the performance differences because EclipseMR does not share any component with Hadoop software stack. Moreover, Hadoop and Spark are full featured frameworks that provide various generic functionalities that are usually followed by significant overhead. For an example, Hadoop tasks run in Yarn containers and each Yarn container spends more than 7 seconds for initialization and authentication [78]. This overhead becomes significant because the container initialization and authentication repeats for every task. I.e., Hadoop spends 7 seconds for every 128 MB block [79]. Compared to Hadoop and Spark, EclipseMR is a lightweight prototype framework that does not provide any other functionalities than what we present in this paper.

We use the default *fair* scheduling in Hadoop, and the delay scheduling in Spark. Again we submit a single application that accesses 250 GB datasets (or 15 GB datasets for **page rank**) at a time after emptying the OS buffer cache and distributed in-memory caches for non-iterative MapReduce applications. For **page rank**, **kmeans**, and **logistic regression** applications, we enable distributed in-memory caches and set the size of the cache to 1 GB per server.

Figure 6 shows the normalized execution time to the slowest result. For the non-iterative MapReduce applications, Spark often shows slightly worse performance than Hadoop, which we believe is because Spark is specifically tailored for iterative jobs such as **page rank**, **kmeans**, and **logistic regression**, not for non-iterative ETL jobs such as **inverted index**. For non-

iterative jobs, all three frameworks do not benefit from caching. Therefore, the performance differences are mainly caused by scheduling decisions.

The performance of `sort` in Figure 6 shows how efficiently each framework performs the shuffle phase. Spark is known to perform worse than Hadoop for `sort`, and our experiments also confirm it. Spark claims it has improved `sort` since version 1.1, but our experiments with version 1.6 show that Spark is still outperformed by Hadoop and EclipseMR.

For iterative applications, we set the number of iterations to 5 for the `kmeans` application, 2 for the `page rank` application, and 10 for the `logistic regression` application. Since Hadoop is an order of magnitude slower than the other two frameworks, we omit the performance of Hadoop `kmeans` and `logistic regression`.

For `kmeans`, EclipseMR is about 3.5x faster than Spark, and for `logistic regression`, EclipseMR is about 2.5x faster than Spark. Note that our faster C++ implementations of `kmeans` and `logistic regression` contributed to the performance improvement, but there are other performance factors that are not out of scope of this research, i.e., Java heap management, container overhead, and some engineering issues that make Spark tasks unstable also need to be investigated.

For `page rank` application, Spark is about 15% faster than EclipseMR. This is because the size of the input file in `page rank` is small and our cluster has a large enough number of slots to run all the mappers concurrently. So, there's no load balancing issues. Moreover, `page rank` generates a very large output for each iteration; the size of iteration outputs in `page rank` is often similar to that of input data. While Spark does not store the intermediate outputs in file systems, EclipseMR writes the large iteration outputs to the persistent DHT file systems to provide fault tolerance. Therefore, if the size of intermediate results is large, the performance gap between EclipseMR and Spark decreases and EclipseMR is outperformed by Spark.

### 3.3.3 Iterative Applications

In the experiments shown in Figure 7 we further analyze the performance of EclipseMR and Spark for iterative applications - `k-means`, `logistic regression`, and `page rank`. Spark runs the first iteration of the iterative applications much slower than subsequent iterations because it constructs RDDs that can be used by subsequent iterations. For subsequent iterations of `kmeans` and `logistic regression`, EclipseMR runs 3x faster than Spark because it does not wait to be scheduled on the servers that has the iteration outputs in their caches, but it immediately starts running in a remote server and accesses remote cached data.

Similar to `kmeans` and `logistic regression`, `page rank` also runs subsequent iterations faster than the first iteration by taking advantage of input data caching. Unlike `kmeans`, EclipseMR is outperformed by Spark for subsequent `page rank` iterations mainly because EclipseMR writes large iteration outputs to the DHT file system. However, even if EclipseMR writes to slow disks, EclipseMR is at most 30% slower than Spark. With a small 30% IO overhead, EclipseMR can restart from the iteration if system crashes.

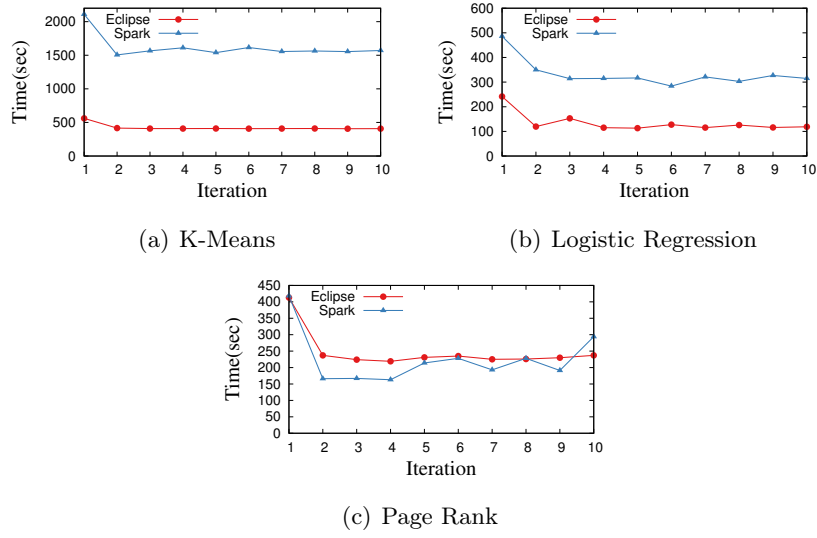


Figure 7: *Execution Time of Iterative Jobs*

Note that Spark runs **page rank** slower than EclipseMR in the last iteration because Spark writes its final outputs to disk storage. The last iterations of **kmeans** and **logistic regression** are not slower than the previous iterations because the outputs of these applications are not as large as **page rank**.

## IV Index Structures for the Persistent Memory

The fundamental challenge in designing a data structure for the persistent memory is to guarantee the failure-atomicity during the transition between one consistent state to the other. It is necessary to rely on the fine-grained memory management scheme with memory fence instruction and cache-line flush instruction. However, it is desirable to minimize the number of these expensive instructions as they hinder the optimization of out-of-order execution and cause cache invalidation.

### 4.1 $B^3$ -Tree: Byte-addressable Binary B-tree

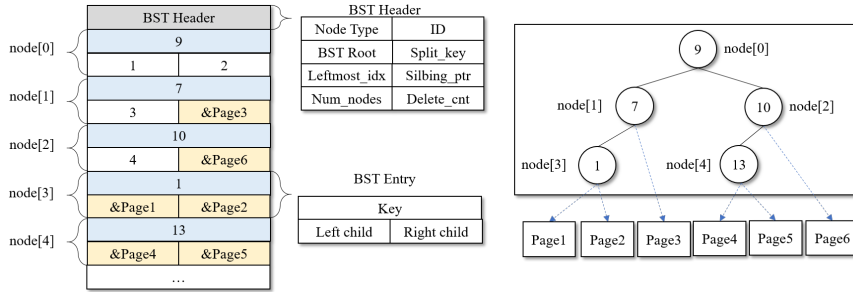


Figure 8: Page Structure of  $B^3$ -tree

#### 4.1.1 Node Structure of $B^3$ -tree

$B^3$ -tree is a hybrid index that combines the positive properties of binary search tree (byte-addressability) and balanced hierarchical B-tree (cache locality).  $B^3$ -tree is a self-balancing binary tree index but its rebalancing operation is similar to that of B-trees rather than the rotations of AVL-tree or red-black tree.  $B^3$ -tree groups a set of nearby BST nodes and stores them in a single  $B^3$ -tree node. To distinguish internal BST nodes from external  $B^3$ -tree nodes, we refer to external  $B^3$ -tree node as  $B^3$ -tree page and internal binary nodes as  $BST$  nodes hereafter.

Figure 8 illustrates the structure of  $B^3$ -tree page. As in B-tree variants,  $B^3$ -tree is an  $n$ -ary tree, i.e., each tree page has multiple pointers to child pages and the ordering of child pages are managed by a BST. As shown in Figure 8, each  $B^3$ -tree page consists of a header and an array of keys and child pointers.

The header stores three fields; i) the *root offset* stores the offset of the root BST node, ii) the *page type* field indicates whether the current page is a leaf  $B^3$ -tree page or an internal  $B^3$ -tree page, iii) and the sibling pointer points to its external sibling  $B^3$ -tree page, which we will describe in Section 4.2.2.1.



The array of keys and child pointers stores the binary representation of sorted keys and child page pointers. In legacy BST, a child pointer is a memory address of another BST node. However, in our  $B^3$ -tree page, a child pointer is either the memory address of an external  $B^3$ -tree page or the index of a binary node in the same  $B^3$ -tree page. For example, `node[0]` in Figure 8 has two BST child nodes. Thus, its child pointers are not the memory address but the index of BST nodes. In legacy BST, a tree node may have a single child. However, in our  $B^3$ -tree, we note that all BST nodes have two children because BSTS nodes are created only when a  $B^3$ -tree page splits. For example, if **Page 3** splits, a new BST node is created that point to **Page 3** and a new split page. And the new BST node is pointed by `node[1]`.

## 4.2 Failure-Atomic $B^3$ -Tree Node Update

In this section, we discuss how  $B^3$ -tree achieves failure-atomicity for a single  $B^3$ -tree page update. For multiple page updates triggered by page split or merge, we defer our discussion to Section 4.2.2.1 and 4.2.4.

### 4.2.1 Failure-Atomic Insertion

An insertion into a BST requires a single atomic 8-byte write operation since a new key is always added to a leaf node, which can be done by a single pointer update. In such a sense, BST is failure-atomic and write-optimal for insert operations. Algorithm 1 shows the insertion algorithm of BST in  $B^3$ -tree.

First, we check if the current  $B^3$ -tree page has an available space for a new BST node (line 1–6). If not, we check if there is any BST node that is not pointed by the BST. If so, we garbage collect it to make a space. If the current page is full, we split. If we found a space, we write the new BST node to the found space (line 7–12). Even if we write the new BST node in the `node` array and increase the number of BST nodes, the new BST node is not exposed to other transactions unless it is pointed by its parent BST node. To find the parent BST node, we traverse the BST from root node (line 13). Depending on the key, we add the new BST node as either a left or right child of the parent BST node according to legacy BST insertion algorithm (line 14–19).

Let us now discuss how  $B^3$ -trees tolerates various failures that can occur during the insertion algorithm that we described above. The ordering of these memory writes must be preserved for failure-atomicity. Therefore, we call *persist()* function between each phase, which calls a memory barrier and a cache line flush instruction.

First, suppose a system crashes while we are finding an available space for a new BST node (line 1–6). The failure will not result in inconsistency because we have not made any modifications to the BST. Next, suppose a system crashes while a new BST node is being written



---

**Algorithm 1** InsertBSTNode(key, left, right)
 

---

```

1: if count == MAX_COUNT then
2:   count = lazyDefragmentation(this);
3: end if
4: if count == MAX_COUNT then
5:   return SPLIT_THIS_NODE;
6: else
7:   idx = count;
8:   node[idx].key = key;
9:   node[idx].left = left;
10:  node[idx].right = right;
11:  persist(&node[idx]);
12:  count++;
13:  persist(&count);
14:  parent = searchParent(key);
15:  if key < node[parent].key then
16:    node[parent].left = idx;
17:    persist(&node[parent].left);
18:  else
19:    node[parent].right = idx;
20:    persist(&node[parent].right);
21:  end if
22: end if

```

---

(line 7–9). Even if cache line flush instruction is not explicitly called, the dirty cache lines can be flushed via cache replacement mechanisms. However, it still does not hurt consistency as the new BST node has not been added to the BST yet. Moreover, we have not increased the counter variable. Hence, the partially written BST node will be overwritten by a subsequent write transaction.

In line 11, we increase the number of BST nodes. This operation must not be reordered with the previous operations. Thus, we call *persist()* function to make sure the new BST node is flushed (line 11–12). If a system crashes after the increased counter is flushed to persistent memory, the new BST node will be a memory leak, which we refer to as (*dead node*), since it is not pointed by the BST. That is, the increased counter will waste memory space. However, we note that it does not violate the invariants of index. In  $B^3$ -tree, we reclaim such a dead space in a lazy manner. That is, when a  $B^3$ -tree page overflows, we scan the array of BST nodes and check if there are dead nodes that can not be reached from the the root node (line 2). If we find one, we use the dead space for the new BST node. If there is no such a dead node, we split the page (line 4). With such a *lazy defragmentation scheme*, we can reduce the overhead of

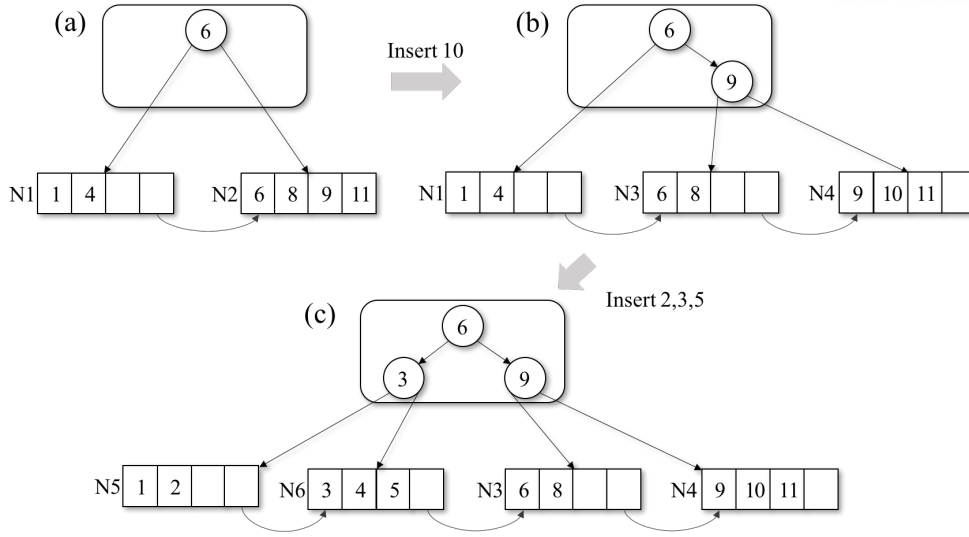


Figure 9: Insertion into  $B^3$ -tree Page

searching an available free space for each insertion. We note that writing a child pointer (offset) in the parent BST node behaves as a commit mark of insert transaction (line 14–19). Since the child pointer is either 8 bytes, the child pointer can be atomically updated. Hence, an insertion into a  $B^3$ -tree node is failure-atomic. If the child pointer is not flushed, the new BST node is not accessible but we note that it does not violate the invariants of index.

#### 4.2.1.1 Failure-Atomic Deletion

Deletion of a tree node in legacy BST is slightly more complicated than insertion because the deletion of an internal BST node updates multiple child pointers.

First, we describe how we make the deletion of a BST node in an internal  $B^3$ -tree page failure-atomic. In internal  $B^3$ -tree page, all BST nodes have two children as shown in Figure 9. A child of BST node can be either another BST node in the same page or another  $B^3$ -tree page. Note that the deletion of a BST node is triggered when we delete a  $B^3$ -tree page. That is, we do not delete an internal BST node that points to two BST nodes. If a  $B^3$ -tree page is deleted, the parent BST node is no longer necessary. Therefore, we simply delete the parent BST node and make the grand parent point to the other child of the parent BST instead. As such, internal  $B^3$ -tree page does not need complicated rotation operations.

Suppose we merge  $B^3$ -tree page N1 and N3 In Figure 9(b). As we remove  $B^3$ -tree page N1, its parent BST node 6 is removed from the current page. Since the BST node 6 has a BST node as its right child, and the parent of BST node 6 needs to point to the right child BST node. But in the example, the BST node 6 is the root node. Hence, we set the root offset in the current  $B^3$ -tree page to be BST child node 9.

Now, suppose we delete BST node 9 as we remove  $B^3$ -tree page N3. Similar to the previous

case, we make the parent of the BST node to be deleted point to the child of the removed BST node. Hence, in the example, BST node 6 point to N4. Note that, in our  $B^3$ -tree, rotation operation is not necessary and a single pointer update completes a delete operation. Therefore, delete operation in  $B^3$ -tree is failure-atomic.

Next, if a leaf page of  $B^3$ -tree stores key-value records as a BST instead of an array, deleting a key-value record will remove an internal BST node, which may require updates to multiple pointers. That is, if the key-value record to be deleted has two sub-trees, we need non-failure atomic rotation. To avoid this problem, we can use an additional metadata such as 'deleted' flag as in NV-tree, FP-tree, and wB-tree, or use a sorted array using the Failure-Atomic Shift (FAST) algorithm of FAST and FAIR B-tree.

The detail deletion algorithm is shown in Algorithm 2. We use the same example depicted in Figure 9 to walk through the deletion algorithm. i) The simplest case is when we merge two pages that are pointed by the same BST node (line 9–12). In this case, we make their grand parent BST node point to the merged page. I.e., if we merge page  $N5$  and  $N6$  shown in Figure 9(c), we create a new page  $N1$ , copy records from  $N5$  and  $N6$  to  $N1$ , and set the left child of BST node (6) to the address of the new page  $N1$ . We note that replacing the left child of BST node (6) will behave as a commit mark of the deletion operation, and it can be done via a single 8-byte atomic write, hence it is failure-atomic. ii) More complicated case is when we merge two pages that are pointed by different BST nodes. Due to the pairwise split and merge algorithm, the parent of one of the pages must have the other page as its leftmost or rightmost child. Suppose we merge a left child page  $N1$  and the right sub-tree's leftmost page  $N3$  shown in Figure 9(b). In this case, we delete the parent BST node of the left page (BST node (6) in the example) from the BST (line 13–21). If the parent node is the root node, we make its right child node a new root node ((9) in the example). Otherwise, we make the grand parent node point to the right child node.

#### 4.2.1.2 Defragmentation

Note that the deletion algorithm of  $B^3$ -tree may leave a hole in the BST nodes array. Such a fragmentation problem degrades the page utilization and makes a search query access more cache lines. To solve this problem,  $B^3$ -tree performs copy-on-write in a lazy manner. That is, we allocate a new page and copy valid BST nodes from the fragmented page to the new page if there is no more available space in the page. During defragmentation, we not only delete invalid BST nodes but also reorganize the tree structure to build a complete binary search tree, thereby shortening the tree height. Even if a system crashes during defragmentation, the failure does not affect the correctness of the tree as the changes are made only in the new copy-on-write page. After we finish the defragmentation, we flush the page and replace the original page by updating pointers in a particular order that we will describe in Section 4.2.2. Note that this lazy defragmentation might hurt the response time of an insertion query that triggers the

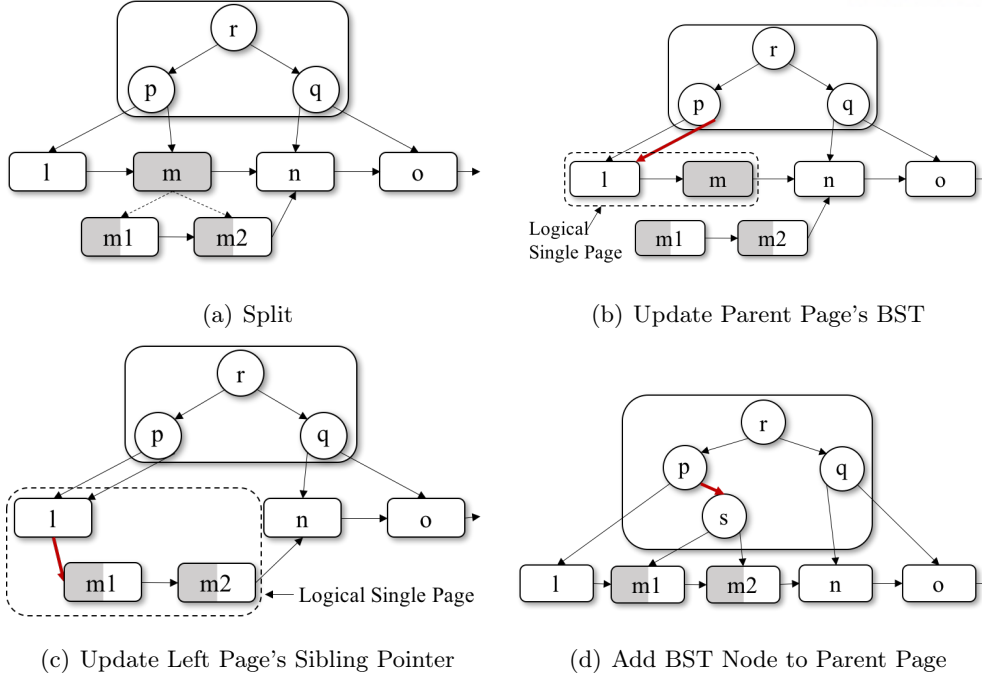


Figure 10: Failure-Atomic Page Split in  $B^3$ -tree

defragmentation process. However, subsequent queries benefit from a fewer number of cache line accesses due to the balanced tree height. Note that there exist self-balancing binary tree structures such as AVL-tree and T-tree. But all of these self-balancing tree structures perform rotation operations to balance the height. However, as we described earlier, rotation operations are not failure-atomic. Hence, we trade off the balanced tree height for failure-atomicity. But the degree of skewness in BSTs are limited by the size of  $B^3$ -tree page and the skewed BSTs regain the balance once in a while via the defragmentation mechanism. As we will show in Section 5.3, improving the page utilization and balancing the tree height via the opportunistic defragmentation help the overall indexing performance.

#### 4.2.2 Failure-Atomic Page Split and Merge

Insertions and deletions often result in page overflows and underflows, which requires  $B^3$ -tree pages to split and merge respectively. Since splitting and merging modify multiple pages to balance the tree height. However, multiple pages cannot be updated atomically. Therefore, legacy B-trees use expensive logging methods, which duplicate dirty pages and increase the write traffic.

Recently, several studies have been conducted to reduce or eliminate the logging overhead by employing byte-addressable persistent memory [37, 41, 80]. NV-tree [37] and FPtree [41] proposed *selective persistence*, which stores internal tree nodes in volatile DRAM but leaf nodes in persistent memory. If a system crashes, internal tree nodes can be reconstructed from leaf

nodes in a bottom-up fashion, hence logging is not necessary for internal nodes. However, such selective persistence is far from satisfactory because reconstruction of a large tree structure is very inexpensive and it makes instant recovery almost impossible [80]. Instead, Hwang et al. [80] proposed the *Failure-Atomic In-place Rebalancing* (FAIR) algorithm that eliminates the necessity of logging and performs in-place rebalancing operations. FAIR algorithm modifies the structure of FAST and FAIR B-tree in a predefined specific order so that read transactions can be aware of the ordering and ignore transient inconsistent tree structures.

For  $B^3$ -tree, we propose a variant of the in-place rebalancing algorithm, i.e., we make  $B^3$ -tree split or merge in a particular order so that read transactions can tolerate transient inconsistent tree structures. Our rebalancing algorithm is different from the FAIR algorithm of FAST and FAIR B-tree in a sense that rebalancing of  $B^3$ -tree structures updates BST structures in the parent page whereas FAST and FAIR B-tree performs shift operations in a sorted array. As in B-link tree [81], every  $B^3$ -tree page has a *sibling pointer* so that all child pages of the same parent page can be managed as a linked list. When a page overflows, we allocate two new pages, connect them via a sibling pointer, redistribute the entries from the overflow page into two new pages, and replace the overflow page with the new pages. That is, the sibling pointer in the new left page is used to combine the left page and the right page so that they become a logical single page until their parent page adds a child pointer to the right page. When two pages merge, we merge them into a newly allocated page and replaces the underflow pages with the new one.

#### 4.2.2.1 Page Split

Figure 21 illustrates the page split algorithm of  $B^3$ -tree. Suppose an insertion causes the page  $m$  overflows. First, i) we allocate two new pages ( $m1$  and  $m2$ ), and construct a balanced complete binary search tree in each page using a half of page  $m$ 's key-value entries. Next, we update the sibling pointers of page  $m1$  and  $m2$  as shown in Figure 21(a). We note that the two new pages are not added to the tree structure yet because no existing pages in  $B^3$ -tree has stored the addresses of the new pages, so the two new pages can not be accessed by other transactions. Second, ii) we replace the pointer to the overflowing page in the parent page's BST with the left sibling of overflowing page. In the example, as shown in Figure 21(b), the BST node  $p$ 's left and right child pointers both point to page  $l$ . Although we removed the pointer to the overflowing page  $m$ , we can still access the page  $m$  if we consider sibling pages  $l$  and  $m$  as a logical single page. We can make transactions follow the right sibling pointer if the parent BST node has the same left and right child pointers. Or alternatively, we can make transactions follow the right sibling pointer if a given search key is greater than the largest key in the page. Next, iii) we set the sibling pointer of page  $l$  to the address of page  $m1$ . By making the left sibling page point to the new split pages, we can atomically remove the overflowing page  $m$  and add the two new split pages  $m1$  and  $m2$ . We note that three pages  $l$ ,  $m1$ , and  $m2$  are a single logical page as shown in Figure 10(c). Finally, iv) we add a new BST node for split pages in the parent BST.

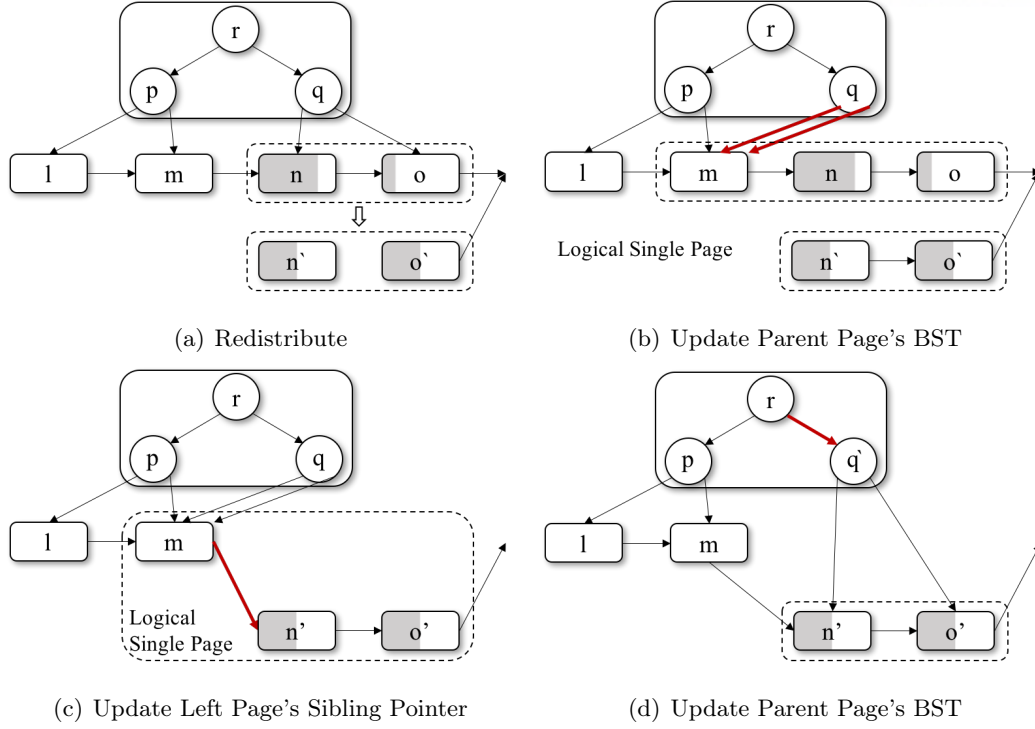


Figure 11: Failure-Atomic Redistribution

In the example shown in Figure 10(c), we create a BST node  $s$  that has page  $m1$  and  $m2$  as its left and right child. Then, we atomically replace the right child pointer of BST node  $p$  with the address of the new BST node  $s$  in the parent page. After completing the split, we insert the data that caused the overflow. Algorithm 3 describes the details of  $B^3$ -tree split algorithm.

#### 4.2.3 Crash Recovery During Page Split

While a page is splitting, various system failures can occur at any time. Suppose a system crashes while allocating two new pages or copying a half of entries to each new page as shown in Figure 21(a). Since the original overflowing page is not altered, recovery is trivial. That is, we can simply ignore and deallocate the newly allocated pages. We note that memory allocators for persistent memory must guarantee the failure atomicity for memory allocation and deallocation, as was proposed in NVWAL [27].

If a system crashes after we set the parent BST node's right child pointer to the left child pointer as shown in Figure 21(b) and 10(c), the recovery process, which scans the entire tree structure, will detect whether a BST node has identical pointers. If it does, the recovery process checks if the right sibling page and the sibling of right sibling page are pointed by the parent BST. If not, we fix the BST so that the child pages can be directly accessed. We note that the recovery process does not need a separate log file. In addition, read transactions always succeed finding a key even if a node split has not completed. For example, if a query looking for a key

in  $m2$  is submitted when a  $B^3$ -tree is in the state shown in Figure 10(c), the query will access  $l$  first,  $m1$  next, and then  $m2$ .

#### 4.2.4 Failure-Atomic Redistribution

If the utilization of a page drops below a threshold value (e.g., 50% in our implementation), B-tree variants perform redistribution, that is, the underutilized page borrows some entries from its sibling page. If the redistribution makes the sibling page also underutilized, the underutilized page and the sibling page are merged.

To perform redistribution or merge in a failure-atomic way,  $B^3$ -tree again performs in-place updates using logical pages. Page merge algorithm is similar to the page split algorithm, but the order of operations is reversed. First, we remove the pointer to the right sibling page from the parent page's BST as shown in Figure 10(c). This will make  $l$ ,  $m1$ , and  $m2$  in the example a logical single page. Then, we allocate a new page ( $m$  in the example) and copy entries from underutilized pages ( $m1$  and  $m2$ ) to the new page. Note that we sort the keys and construct a complete BST in the merged page. When the merged page is ready, we atomically update the sibling pointer of the left sibling page ( $l$  in the example) as shown in Figure 21(b). Finally, we update the parent page's BST so that the merged page ( $m$ ) is pointed by the parent BST.

Redistribution is similar to the merge algorithm. We use the example depicted in Figure 11 to walk through the detailed workings of failure-atomic redistribution. First, i) we allocate two new pages ( $n'$  and  $o'$ ) and redistribute entries from the two underutilized pages, as shown in Figure 11(a). Next, ii) we remove the pointers to underutilized pages ( $n'$  and  $o'$ ) from the parent BST, as shown in Figure 11(b). Updating the two child pointers of  $q$  does not have to be atomic. Making the left child pointer point to page  $m$  will make page  $m$  and  $n$  a logical single page. Then, making the right child pointer point to page  $m$  will make three pages -  $m$ ,  $n$ , and  $o$  a logical single page. We note that these operations do not affect the invariants of the index, hence they are failure-atomic. In the next step, iii) we update the sibling pointer of the leftmost page in the logical single page ( $m$  in the example), as depicted in Figure 11(c)), so that the underutilized pages are replaced by the redistributed pages -  $n'$  and  $o'$ . Finally, iv) we replace BST node ( $q$ ) in the parent page with a new BST node ( $q'$  in the example), as shown in Figure 11(d). Updating the right child pointer of  $r$  via atomic 8-byte write operation completes the redistribution.

##### 4.2.4.1 Rebalancing BST

Each page in  $B^3$ -tree is represented as a BST. Although there are numerous byte-addressable in-memory indexing structures such as AVL-tree and T-tree that keep the height of sub-trees balanced, we propose to use the simple BST because it does not require tree rotation operations.



A tree rotation modifies multiple nodes and calls multiple cache line flushes. However, BST always add a new tree node as a child of a leaf node. Therefore, a single failure atomic cache line flush is sufficient to insert a new node. However, there is a trade-off. Although BST is good for failure-atomic insertion, BST suffers from the notorious *skew* problem. The number of BST nodes in  $B^3$ -trees is bounded by the page size. When the page size is 4 KB, maximum 127 8-byte keys can be indexed in our implementation. Therefore, the height of BST can be as high as 127 in the worst case.

$B^3$ -tree bounds the number of comparisons even if BSTs are completely skewed because BSTs are only used for internal page representation of perfectly balanced B-trees. However, although the skewness of  $B^3$ -tree page is bounded, skewed BSTs degrade the page access performance in terms not only of search but also of insertion as well. To mitigate the skew problem of BST, we make  $B^3$ -tree reconstruct a complete BST when a page is split or when sibling pages merge. Note that splitting or merging modifies a large portion of pages. Hence, we let  $B^3$ -tree perform copy-on-write instead of in-place updates even on byte-addressable persistent memory. As such, the overhead of calling a large number of `clflush` when rebalancing BSTs is masked by the overhead of copy-on-write.

To convert a skewed BST into a complete BST, we in-order traverse the skewed BST and store the result in the BST node array. This optimization takes  $O(n)$  time where  $n$  is the number of binary tree nodes. This rebalancing optimization not only shortens the tree height but also improves cache line locality.

#### 4.2.5 Concurrency Control

With the growing number of cores in modern computer architectures, non-blocking access to concurrent data structures is drawing more attention in the community. To improve concurrent access to the index, FP-tree [41] employs the hardware transactional memory (HTM), and FAST and FAIR B-tree [80] eliminates the necessity of read locks by making read transactions tolerate transient inconsistent status.

Similar to FAST and FAIR B-tree, a sequence of 8 byte store instructions used by  $B^3$ -tree also guarantees the consistency of data structures as described earlier. Hence, access to the shared  $B^3$ -tree does not have to be serialized and enables lock-free search as in FAST and FAIR B-tree. That is, even if a write thread fails or a system crashes while making changes to  $B^3$ -tree, no read thread will ever access inconsistent tree nodes because every single store instruction in  $B^3$ -tree does not affect the invariants of index and guarantees correct search results. Therefore, read operations do not have to wait for write transactions to finish updates and to release exclusive locks.

On the other hand, if multiple write threads update the same tree nodes simultaneously,  $B^3$ -tree suffers from write-write conflicts. Suppose a write thread is about to add a BST node to a leaf node and another thread is trying to delete the leaf node at the same time. If the insertion succeeds right before the leaf node is deleted, the newly inserted node will not be a part of



the index, which leaves the index inconsistent. Therefore,  $B^3$ -tree does not allow concurrent modifications to the same page. A write thread must acquire an exclusive lock to modify a page.

In various applications including enterprise database systems, read transactions are much more popular than write transactions. Hence, lock-free search alone can significantly improve the query processing throughput even if write threads still need exclusive locks.

## 4.3 Evaluation

### 4.3.1 Experimental Setup

We run experiments on a workstation that has four Intel Xeon Haswell-EX E7-4809 v3 processors (8 cores, 2.0GHz, 8x32KB instruction cache, 8x32 KB data cache, 8x256 KB L2 cache, and 20 MB L3 cache) and 64 GB of DDR3 DRAM.

Byte-addressable persistent main memory is not commercially available yet. Thus, we emulate persistent memory using *Quartz* [82, 83], a DRAM-based PM latency emulator [82]. Quartz consists of a kernel module and a user-mode library that models application-perceived latency by injecting stall cycles into each predefined time interval, called *epoch*. We configure minimum and maximum epochs to 5 nsec and 10 nsec for all our experiments. We adjust read latency of PM from 300 nsec, which is the minimum latency that we can configure in our testbed environment. We set the maximum PM latency to 900 nsec because current projections indicate that latencies of most PM technologies will be higher than DRAM but no higher than an order of magnitude [38]. To emulate write latency, we inject stall cycles after each `clflush` instructions, as was done by [84, 27, 33, 39, 85]. Note that PM write latency is often hidden by CPU cache. Hence, we do not add the software delay to `store` instructions. As for the PM bandwidth, we assume that PM bandwidth is no different from that of DRAM since Quartz does not allow us to emulate both latency and bandwidth at the same time.

We compare the performance of  $B^3$ -tree against `wB+-tree` and `FPTree`. Unlike  $B^3$ -tree and `wB+-tree`, `FPTree` is not a persistent index in a strict sense because it stores internal tree pages in volatile DRAM. Thereby, it is less sensitive to a PM latency. Unlike `FPTree`,  $B^3$ -tree and `wB+-tree` store all tree pages in PM. Hence, their failure-atomic page update algorithms make instant recovery possible at the cost of slightly slow access to internal pages.

### 4.3.2 Performance Effect of Page Size

In the experiments shown in Figure 12, we insert 10 million keys in a random order while we vary the page size of  $B^3$ -tree. We assume the latency of PM is the same with that of DRAM and compare the insertion performance of  $B^3$ -tree against `wB+-tree`. Figure 12(a) shows the insertion time of  $B^3$ -tree is 45% of that of `wB+-tree` and 41% of that of `wB+-tree` without a bitmap. As we increase the page size, the insertion time of  $B^3$ -tree slightly decreases because of

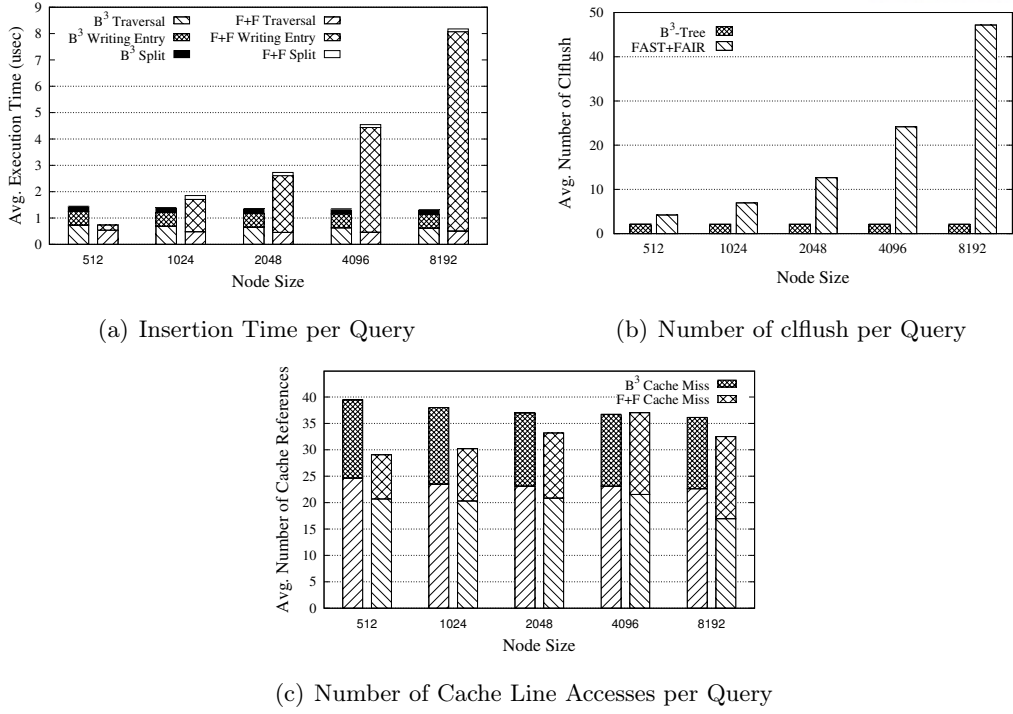


Figure 12: Insertion Performance with Varying Page Sizes

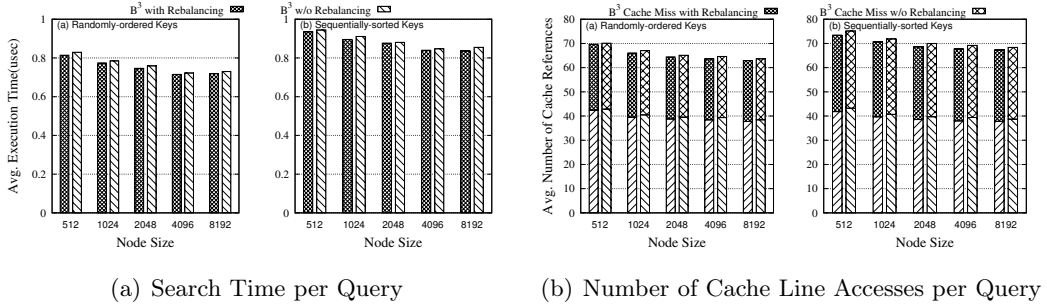


Figure 13: Search Performance: Balanced Trees vs Skewed Trees

a fewer number of split operations, but not significantly because our rebalancing optimization restructures BSTs in such a way that a query visits a similar number of BST nodes ( $O(\log k)$ , where  $k$  is the number of BST nodes determined by page size).

On the other hand, the page size of wB+-trees is not adjustable. That is, wB+-tree with a bitmap can have up to 63 keys (1 KB page size) and wB+-tree without a bitmap can have only 8 keys. Figure 12(b) shows the average number of `clflush` instructions. Because of the bitmap, wB+-tree calls `clflush` instructions at least four times for normal insertions and dozens of `clflush` instructions for split and merge operations due to logging. Overall, wB+-tree with a bitmap and without a bitmap invoke about seven and six `clflush` instructions respectively, but B<sup>3</sup>-tree calls only two `clflush` instructions per insertion.

Figure 12(c) shows the number of CPU cache references and the number of LLC misses per query. B<sup>3</sup>-tree shows a slightly larger number of cache line access than wB+-trees with a

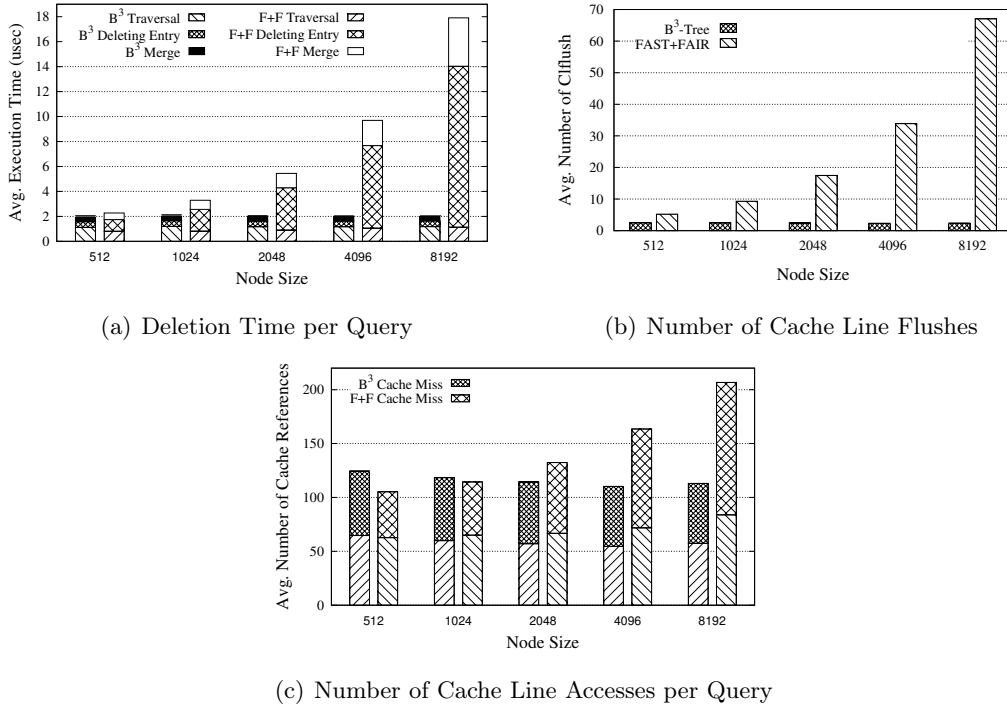


Figure 14: Deletion Performance with Varying Page Sizes

bitmap, and it shows a higher cache miss ratio than wB+-trees. Even though the cache miss ratio of  $B^3$ -tree is greater than wB+-trees,  $B^3$ -tree shows better insertion performance because of its minimal metadata management and a small number of cache line flushes.

Figure 13 shows how the rebalancing optimization effectively balances the tree height of BSTs in  $B^3$ -tree and improves the search performance. The performance of BST is often affected by the distribution of keys and an insertion order. If we insert keys in a sorted order, BSTs is often completely skewed and the tree height will be the number of BST nodes ( $O(n)$ ), which is the worst case in BST. Therefore, we evaluate the performance of  $B^3$ -trees using two types of workloads - monotonically increasing keys and random keys.

If we disable the rebalancing optimization and insert monotonically increasing keys, the height of BST grows linearly as we increase the page size. However, Figure 13(b) shows that the rebalancing optimization effectively reduces the number of accessed cache lines and the number of LLC misses. As a result, the performance of two different workloads are similar.

Figure 14 shows the deletion performance. Similar to the insertion and search performance, the deletion time of  $B^3$ -tree is also almost independent of page size because of the rebalancing optimization. Figure 14(b) shows that  $B^3$ -tree calls about 2.5 cache line flushes on average. Although the page merge algorithm of  $B^3$ -tree requires multiple cache line flushes, most deletions require only a single cache line flush. Therefore, the average number of cache line flushes is much smaller than that of wB+-tree.

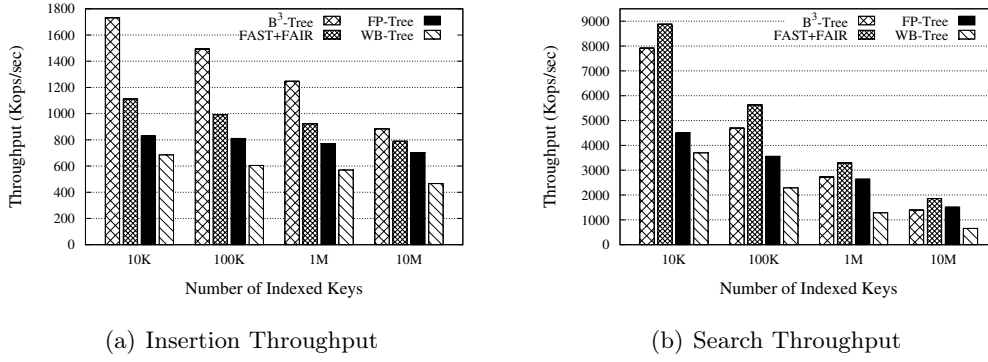


Figure 15: Throughput Comparison with Varying Number of Indexed Data

### 4.3.3 Comparative Performance

In the experiments shown in Figure 15, we insert various numbers of random key-value pairs and measure the insertion and search throughput of persistent indexes. In terms of the insertion throughput shown in Figure 15(a),  $B^3$ -tree consistently outperforms all other indexes. Note that  $B^3$ -tree shows up to 45% higher insertion throughput than FAST and FAIR B-tree, and  $B^3$ -tree shows about 2x higher insertion throughput than wB+-tree. As the index size becomes larger, the performance gap between  $B^3$ -tree and FAST and FAIR B-tree decreases because a page split in  $B^3$ -tree performs the copy-on-write for two new sibling pages in order to rebalance two BSTs, but FAST and FAIR B-tree creates only one sibling page and performs the in-place update for the overflowing page. As a result, FAST and FAIR modifies a fewer number of cache lines than  $B^3$ -tree for split operations. However, note that we run this experiments assuming the PM latency is no different from that of DRAM. As we will show in the next set of experiments, the performance gap between  $B^3$ -tree and FAST and FAIR B-tree widens as we increase the PM latency because of the leaf page update overhead. We also note that the performance gap between  $B^3$ -tree and FPtree also decreases as we index more key-value pairs because FPtree does not have `clflush` overhead when updating internal pages and the number of internal pages increases with a larger index size.

As for the search throughput, shown in Figure 15(b),  $B^3$ -tree is slightly outperformed by FAST and FAIR B-tree because leaf pages of  $B^3$ -tree do not sort the key-value pairs. As a result, the number of comparison operations in  $B^3$ -tree leaf pages is higher than FAST and FAIR B-tree. Interestingly, the search throughput of wB+-tree is higher than that of FPtree when the index size is small. However, as the index size increases, FPtree benefits from faster access to internal pages in DRAM and its search throughput becomes higher than wB+-tree and similar to  $B^3$ -tree. Again, we note that FPtree is not a persistent index because FPtree needs to be reconstructed from scratch when a system crashes.

#### 4.3.4 PM Latency Effect

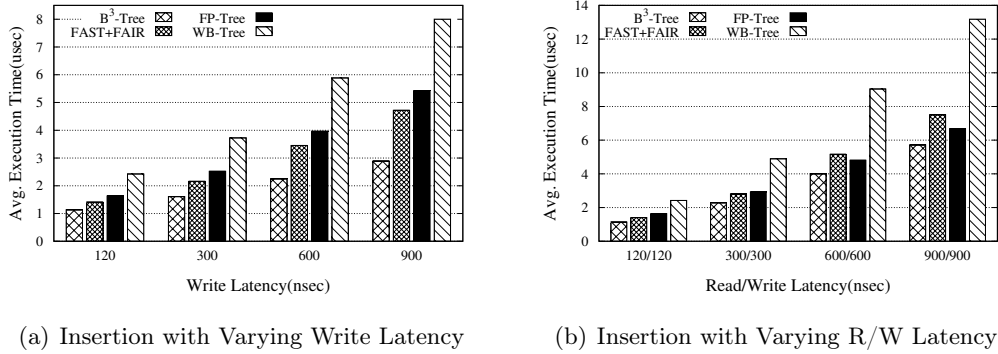


Figure 16: Insertion Performance Comparison (Latency)

Now, we index 10 million random keys and measure the average insertion time with varying the latency of PM. In the experiments shown in Figure 16(a), we set the read latency of PM to that of DRAM but we vary the write latency of PM. As we increase the write latency of PM, FAST and FAIR B-tree suffers from a larger number of cache line flushes caused by shift operations. Therefore, its insertion time increases at a faster rate than that of  $B^3$ -tree. If the write latency of PM is equal to that of DRAM,  $B^3$ -tree and FP-Tree show comparable insertion performance. However, as the write latency increases,  $B^3$ -tree shows about 45% faster performance than FP-Tree although FP-Tree benefits from faster DRAM latency. This is because  $B^3$ -tree does not perform expensive logging and it calls a fewer number of cache line flushes than FP-Tree. However, if we increase both read and write latency, as shown in Figure 16(b), FP-Tree shows a similar insertion performance with  $B^3$ -tree because the logging overhead of FP-Tree is masked by faster access to internal pages.

#### 4.3.5 Concurrency

In the experiments shown in Figure 18 and 19, we evaluate the performance of multi-threaded versions of  $B^3$ -tree, FAST and FAIR B-tree, and FP-Tree. While  $B^3$ -tree and FAST and FAIR B-tree enables lock-free search, FP-Tree employs Intel’s Transactional Synchronization Extension (TSX).

In Figure 18, we measure the insertion and search throughput with varying the number of concurrent threads. We insert 10 million keys into the index that already has 10 million keys. Our testbed machine has 16 vCPUs. Therefore, the speed up from concurrent threads becomes saturated when we run more than 16 threads. We note that  $B^3$ -tree shows higher insertion throughput than FAST and FAIR B-tree when the number of threads is smaller than 16. We note that the performance of  $B^3$ -tree outperforms FP-Tree in terms of both insertion and search throughputs as we increase the number of threads. Although the single-threaded read

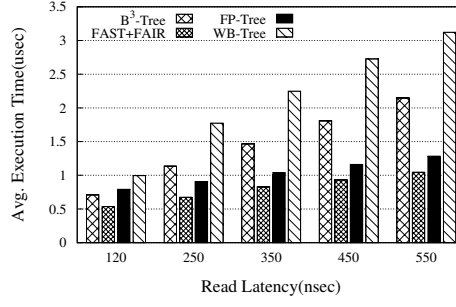


Figure 17: Search Performance with Varying Read Latency

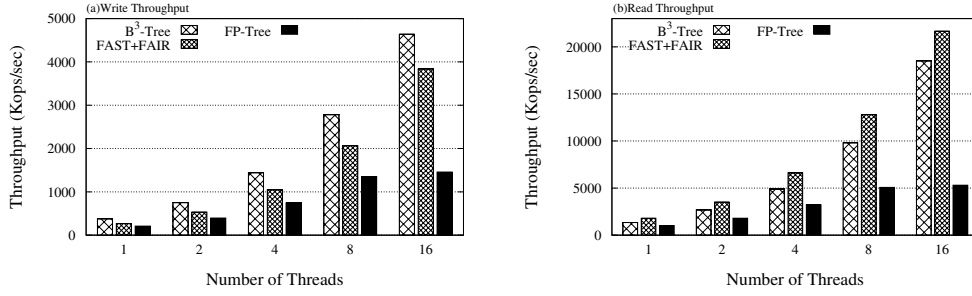


Figure 18: Insertion(a) and Search(b) Throughputs with Varying Number of Threads

performance of  $B^3$ -tree is shown to be lower than FPtree in Figure 17,  $B^3$ -tree takes advantage of a higher level of concurrency than FPtree. Since read threads of  $B^3$ -tree and FAST and FAIR B-tree do not need to acquire read locks, both indexes achieve linear scalability while read threads using FPtree suffer from thread contention.

In the experiment shown in Figure 19, we measure the average execution time of 10 million queries. We vary the number of read and write transactions in order to vary the read/write ratio. As we increase the write ratio, the total execution time increases. However, FPtree suffers the most from the higher write ratio, because write threads in FPtree suspend read threads while the other two indexes allow read transactions to access index without being blocked. Note that if we run 32 threads concurrently, the total execution time of FPtree is 5x longer than that of  $B^3$ -tree.

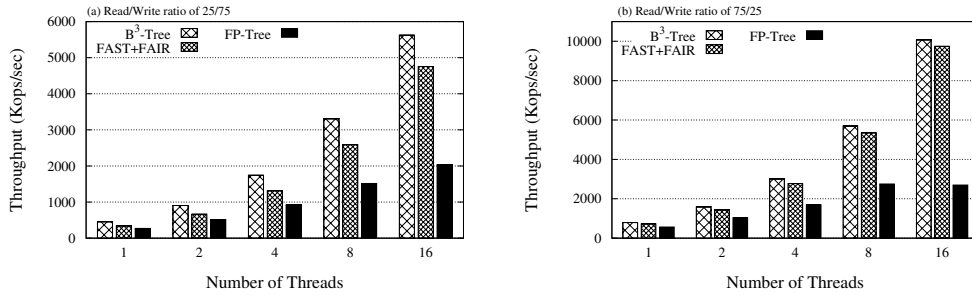


Figure 19: Performance Comparison with 4 Threads(l) and 16 Threads(r) Using Mixed Workload

---

**Algorithm 2** MergeTreePages(leftPage, rightPage)

---

```

1: lp = parentBSTNode(leftPage);
2: rp = parentBSTNode(rightPage);
3: mergedPage = merge(leftPage, rightPage);
4: leftSibling = findLeftSiblingPage(leftPage);
5: lp→swapAndPersist(leftPage, leftSibling);
6: rp→swapAndPersist(rightPage, leftSibling);
7: leftSibling→sibling = mergedPage;
8: persist(&leftSibling→sibling);
9: if lp == rp then
10:    // if two merged pages share the same parent
11:    grandParent = parentBSTNode(lp);
12:    grandParent→swapAndPersist(lp, mergedPage);
13: else if lp.left == leftPage then
14:    rp.left = mergedPage;
15:    persist(&rp.left);
16:    if node→root == lp then
17:        node→root = lp.right;
18:        persist(&node→root);
19:    else
20:        grandParent = parentBSTNode(lp);
21:        grandParent→swapAndPersist(lp, lp.right);
22:    end if
23: else if lp.right == leftPage then
24:    lp.right = mergedPage;
25:    persist(&lp.right);
26:    if node→root == rp then
27:        node→root = rp.left;
28:        persist(&node→root);
29:    else
30:        grandParent = parentBSTNode(rp);
31:        grandParent→swapAndPersist(rp, rp.left);
32:    end if
33: end if
34: node→delete_cnt++;

```

---

---

**Algorithm 3** SplitPage(parent, n)

---

```
1: n1 = alloc();
2: n2 = alloc();
3: m = findMedian(n);
4: copyEntries(n, 0, m, n1);
5: copyEntries(n, m, ∞, n2);
6: n1.sibling = &n2.id;
7: n2.sibling = n.sibling;
8: persist(&n1);
9: persist(&n2);
10: leftSibling = findLeftSiblingPage(node);
11: swapAndPersist(parent, n, leftSibling);
12: leftSibling→sibling = n1;
13: persist(&leftSibling→sibling);
14: InsertBinaryTreeNode(m, n1, n2);
```

---



#### 4.4 Cacheline-Conscious Extendible Hashing

In this section, we present Cacheline-Conscious Extendible Hashing (CCEH), a variant of extendible hashing that overcomes the shortcomings of traditional extendible hashing by guaranteeing the failure-atomicity and reducing the number of cacheline accesses for the benefit of byte-addressable PM.

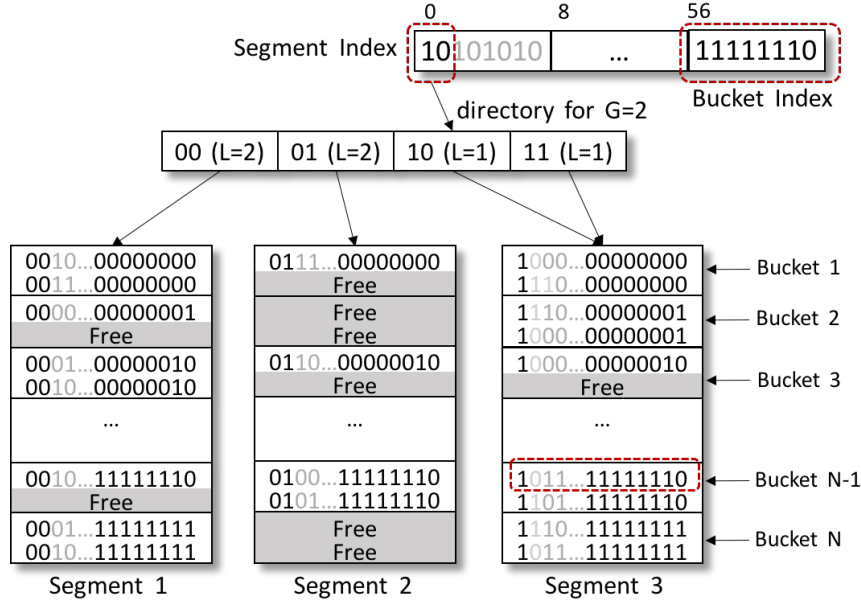


Figure 20: *Cacheline-Conscious Extendible Hashing*

##### 4.4.1 Three Level Structure of CCEH

In byte-addressable PM, the unit of an atomic write is a word but the unit of data transfer between the CPU and memory corresponds to a cacheline. Therefore, the write-optimal size of a hash bucket is a cacheline. However, a cacheline, which is typically 64 bytes, can hold no more than four key-value pairs if the keys and values are word types. Considering that each cacheline sized-bucket needs an 8-byte pointer in the directory, the directory can be the tail wagging the dog, i.e., if each 64-byte bucket is pointed by a single 8-byte directory entry, the directory can be as large as 1/8 of the total bucket size. If multiple directory entries point to the same bucket, the directory size can be even larger. To keep the directory size under control, we can increase the bucket size. However, there is a trade-off between bucket size and lookup performance as increasing bucket size will make lookup performance suffer from the large number of cacheline accesses and failure to exploit cache locality.

In order to strike a balance between the directory size and lookup performance, we propose to use an intermediate layer between the directory and buckets, which we refer to as a *segment*. That is, a segment in CCEH is simply a group of buckets pointed to by the directory. The structure of CCEH is illustrated in Figure 20. To address a bucket in the three level structure,

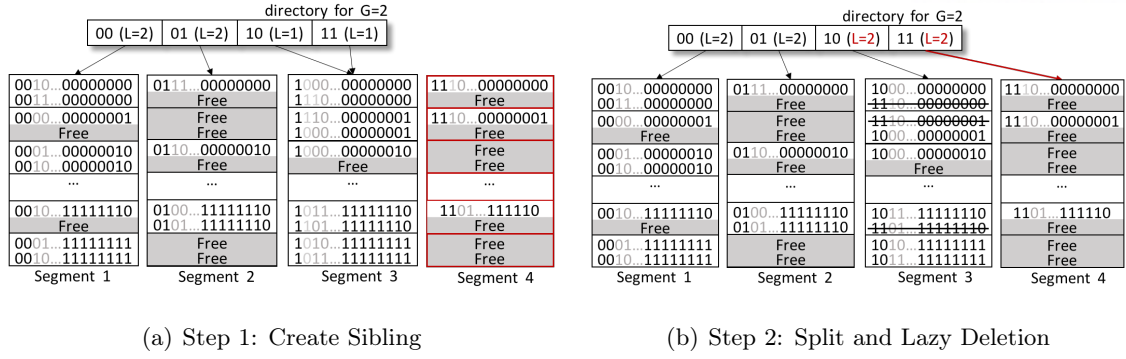


Figure 21: *Failure-Atomic Segment Split Example*

we use the  $G$  bits (which represents the global depth) as a segment index and an additional  $B$  bits (which determines the number of cachelines in a segment) as a bucket index to locate a bucket in a segment.

In the example shown in Figure 20, we assume each bucket can store two records (delimited by the solid lines within the segments in the figure). If we use  $B$  bits as the bucket index, we can decrease the directory size by a factor of  $1/2^B$  ( $1/256$  in the example) compared to when the directory addresses each bucket directly. Note that although the three level structure decreases the directory size, it allows access to a specific bucket (cacheline) without accessing the irrelevant cachelines in the segment.

Continuing the example in Figure 20, suppose the given hash key is  $10101010...11111110_{(2)}$  and we use the least significant byte as the bucket index and the first two leading bits as the segment index since the global depth is 2. We will discuss why we use the leading bits instead of trailing bits as the segment index later in Section 4.4.4. Using the segment index, we can lookup the address of the corresponding segment (Segment 3). With the address of Segment 3 and the bucket index ( $11111110_{(2)}$ ), we can directly locate the address of the bucket containing the search key, i.e.,  $(\&\text{Segment3} + 64 \times 11111110_{(2)})$ . Even with large segments, the requested record can be found by accessing only two cachelines - one for the directory entry and the other for the corresponding bucket (cacheline) in the segment.

#### 4.4.2 Failure-Atomic Segment Split

A split performs a large number of memory operations. As such, a segment split in CCEH cannot be performed by a single atomic instruction. Unlike full-table rehashing that requires a single failure-atomic update of the hash table pointer, extendible hashing is designed to reuse most of the segments and directory entries. Therefore, the segment split algorithm of extendible hashing performs several in-place updates in the directory and copy-on-writes.

In the following, we use the example depicted in Figure 21 to walk through the detailed workings of our proposed failure-atomic segment split algorithm. Suppose we are to insert key  $1010...11111110_{(2)}$ . Segment 3 is chosen as (the local depth is 1 and) the leftmost bit is 1, but

the 255th ( $11111110_{(2)}$ th) bucket in the segment has no free space, i.e., a hash collision occurs. To resolve the hash collision, CCEH allocates a new Segment and copies key-value records not only in the collided bucket of the segment but also in the other buckets of the same segment according to their hash keys. In the example, we allocate a new Segment 4 and copy the records, whose key prefix starts with 11, from Segment 3 to Segment 4. We use the two leading bits because the local depth of Segment 3 will be increased to 2. If the prefix is 10, the record remains in Segment 3, as illustrated in Figure 21(a).

In the next step, we update the directory entry for the new Segment 4 as shown in Figure 21(b). First, (1) the pointer and the local depth for the new bucket are updated. Then, (2) we update the local depth of the segment that we split, Segment 3. I.e., we update the directory entries from right to left. The ordering of these updates must be enforced by inserting `mfence` instruction in between each instruction. Also, we must call `clflush` when it crosses the boundary of cachelines, as was done in FAST and FAIR B-tree [80]. Enforcing the order of these updates is particularly important to guarantee recovery. Note that these three operations cannot be done in an atomic manner. That is, if a system crashes during the segment split, the directory can be recovered to a partially updated inconsistent state. For example, the updated pointer to a new segment is flushed to PM but two local depths are not updated in PM. However, we note that this inconsistency can be easily detected and fixed by a recovery process without explicit logging. We detail our recovery algorithm later in Section 4.4.5.

A potential drawback of our split algorithm for three level CCEH is that a hash collision may split a large segment even if other buckets in the same segment have free space. To improve space utilization and avoid frequent memory allocation, we can employ ad hoc optimizations such as *linear probing* or *cuckoo displacement*. Although these ad hoc optimizations help defer expensive split operations, they increase the number of cacheline accesses and degrade the index lookup performance. Thus, they must be used with care. In modern processors, serial memory accesses to adjacent cachelines benefit from hardware prefetching and memory level parallelism [80]. Therefore, we employ simple linear probing that bounds the number of buckets to probe to four cachelines to leverage memory level parallelism.

Similar to the segment split, a segment merge performs the same operations, but in the reversed order. That is, (1) we migrate the records from the right segment to the left segment, as shown in Figure 21(b). Next, (2) we decrease the local depths and update pointers of the two segments in the directory, as shown in Figure 21(a). Note that we must update these directory entries from left to right, which is the opposite direction to that used for segment splits. This ordering is particularly important for the recovery. Details about the ordering and recovery will be discussed in Section 4.4.5.

### 4.4.3 Lazy Deletion

In legacy extendible hashing, a bucket is atomically cleaned up via a page write after a split such that the bucket does not have migrated records. For failure-atomicity, disk-based extendible

hashing updates the local depth and deletes migrated records with a single page write.

Unlike legacy extendible hashing, CCEH does not delete migrated records from the split segment. As shown in Figure 21(b), even if Segments 3 and 4 have duplicate key-value records, this does no harm. Once the directory entry is updated, queries that search for migrated records will visit the new segment and queries that search for non-migrated records will visit the old segment but they always succeed in finding the search key since the split Segment 3 contains all the key-value records, with some unneeded duplicates.

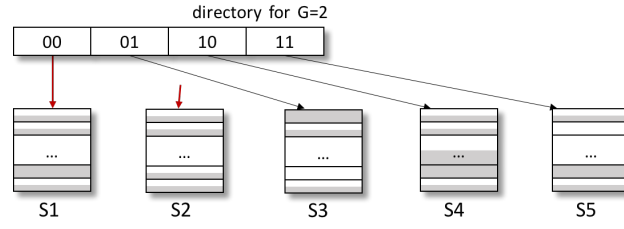
Instead of deleting the migrated records immediately, we propose *lazy deletion*, which helps avoid the expensive copy-on-write and reduce the split overhead. Once we increase the local depth of the split segment in the directory entry, the migrated keys (those crossed-out keys in Figure 21(b)) will be considered invalid by subsequent transactions. Therefore, there is no need to eagerly overwrite migrated records because they will be ignored by read transactions and they can be overwritten by subsequent insert transactions in a lazy manner. For example, if we insert a record whose hash key is  $1010\dots11111110_{(2)}$ , we access the second to last bucket of Segment 3 (in Figure 21(b)) and find the first record's hash key is  $1000\dots11111110_{(2)}$ , which is valid, but the second record's hash key is  $1101\dots11111110_{(2)}$ , which is invalid. Then, the insert transaction replaces the second record with the new record. Since the validness of each record is determined by the local depth, the ordering of updating directory entries must be preserved for consistency and failure-atomicity.

#### 4.4.4 Segment Split and Directory Doubling

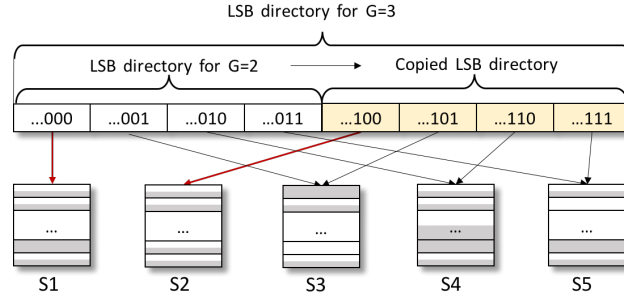
Although storing a large number of buckets in each segment can significantly reduce the directory size, directory doubling is potentially the most expensive operation in large CCEH tables. Suppose the segment pointed to by the first directory entry splits, as shown in Figure 22(a). To accommodate the additional segment, we need to double the size of the directory and make each existing segment referenced by two entries in the new directory. Except for the two new segments, the local depths of existing segments are unmodified and they are all smaller than the new global depth.

For disk-based extendible hashing, it is well known that using the least significant bits (LSB) allows us to reuse the directory file and to reduce the I/O overhead of directory doubling because we can just copy the directory entries as one contiguous block and append it to the end of the file as shown in Figure 22(b). If we use the most significant bits (MSB) for the directory, new directory entries have to be sandwiched in between existing entries, which makes all pages in the directory file dirty.

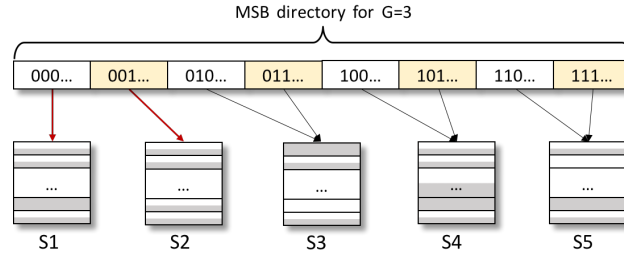
Based on this description, it would seem that making use of the LSB bits would be the natural choice for PM as well. In contrary, however, it turns out when we store the directory in PM, using the most significant bits (MSB) performs better than using the LSB bits. This is because the existing directory entries cannot be reused even if we use LSB since all the directory entries need to be stored in contiguous memory space. That is, when using LSB, we must



(a) Directory with Global Depth=2



(b) Directory Doubling with LSB

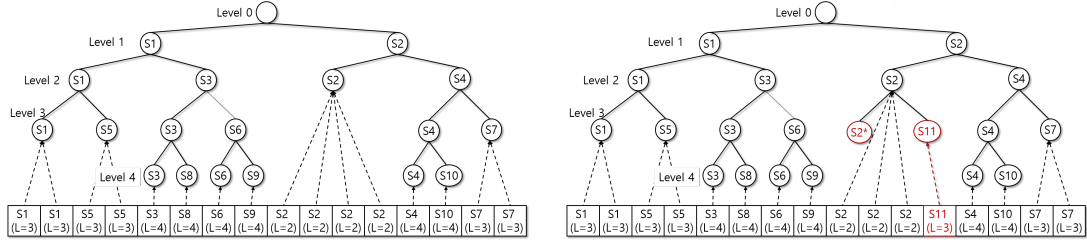


(c) Directory Doubling with MSB

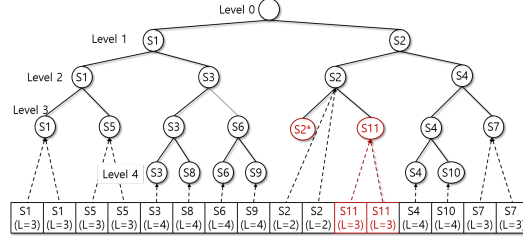
Figure 22: *MSB segment index makes adjacent directory entries be modified together when a segment splits*

allocate twice as much memory as the old directory uses, copy the old directory to the first half as well as to the second half. This is aggravated by the expensive cacheline flushes that are required when doubling the directory. In fact, the overhead of doubling the directory with two `memcpy()` function calls and iterating through a loop to duplicate each directory entry is minimal compared to the overhead of `clflush`. In conclusion, LSB does not help reduce the overhead of enlarging the directory size unlike the directory file on disks.

The main advantage of using MSB over LSB comes from reducing the overhead of segment splits, not from reducing the overhead of directory doubling. If we use MSB for the directory, as shown in Figure 22(c) and 26(a), the directory entries for the same segment will be adjacent to each other such that they benefit from spatial locality. That is, if a segment splits later, multiple directory entries that need to be updated will be adjacent. Therefore, using MSB as segment index reduces the number of cacheline flushes no matter what local depth a split segment has. Even though preserving the spatial locality has little performance effect on reducing



(a) Tree Representation of Segment Split History (b) Split: Update Pointer and Level for new Segment from Right to Left



(c) Split: Increase Level of Split Segment from Right to Left

Figure 23: *Buddy Tree Traversal for Recovery*

the overhead of directory doubling because both MSB and LSB segment index call the same number of `clflush` instructions in batches when doubling the directory, MSB segment index has a positive effect of reducing the overhead of segment splits, which occur much more frequently than the directory doubling. As we will see next, using MSB has another benefit of allowing for easier recovery.

#### 4.4.5 Recovery

Various system failures such as power loss can occur while hash tables are being modified. Here, we present how CCEH achieves failure-atomicity by discussing system failures at each step of the hash table modification process.

Suppose a system crashes when we store a new record into a bucket. First, we store the value and its key next. If the key is of 8 bytes, the key can be atomically stored using the key itself as a commit mark. Even if the key is larger than 8-bytes, we can make use of the leading 8 bytes of the key as a commit mark. For example, suppose the key type is a 32 byte string and we use the MSB bits as the segment index and the least significant byte as the bucket index. We can write the 24 byte suffix first, call `mfence`, store the leading 8 bytes as a commit mark, and call `clflush`. This ordering guarantees that the leading 8 bytes are written after all the other parts of the record have been written. Even if the cacheline is evicted from the CPU cache, partially written records will be ignored because the key is not valid for the segment, i.e., the MSB bits are not a valid segment index. This is the same situation as when our lazy deletion considers a slot with any invalid MSB segment index as free space. Therefore, the partially written records

without the correct leading 8 bytes will be ignored by subsequent transactions. Since all hash tables including CCEH initialize new hash tables or segments when they are first allocated, there is no chance for an invalid key to have a valid MSB segment index by pure luck. To delete a record, we change the leading 8 bytes to make the key invalid for the segment. Therefore, the insertion and deletion operations that do not incur bucket splits are failure-atomic in CCEH.

Making use of the MSB bits as a segment index not only helps reduce the number of cacheline flushes but also makes the recovery process easy. As shown in Figure 23, with the MSB bits, the directory entries allow us to keep track of the segment split history as a binary buddy tree where each node in the tree represents a segment. When a system crashes, we visit directory entries as in binary tree traversal and check their consistency, which can be checked by making use of  $G$  and  $L$ . That is, we use the fact that, as we see in Figure 21, if  $G$  is larger than  $L$  then the directory buddies must point to the same segment, while if  $G$  and  $L$  are equal, then each must point to different segments.

Let us now see how we traverse the directories. Note that the local depth of each segment and the global depth determines the segment's stride in the directory, i.e., how many times the segment appears contiguously in the directory. Since the leftmost directory entry is always mapped to the root node of the buddy tree because of the in-place split algorithm, we first visit the leftmost directory entry and check its buddy entry. In the walking example, the buddy of S1 (directory[0]) is S5 (directory[2]) since its stride is  $2^{G-L} = 2$ . After checking the local depth and pointer of its right buddy, we visit the parent node by decreasing the local depth by one. I.e., S1 in level 2. Now, the stride of S1 in level 2 is  $2^{G-L} = 4$ . Hence, we visit S3 (directory[4]) and check its local depth. Since the local depth S3 is higher (4 in the example), we can figure out that S3 has split twice and its stride is 1. Hence, we visit directory[5] and check its consistency, continuing this check until we find any inconsistency. The pseudo code of this algorithm is shown in Algorithm 4.

Suppose a system crashes while splitting segment S2 in the example. According to the split algorithm we described in Section 4.4.2, we update the directory entries for the split segment from right to left. Say, a system crashes after making directory[11], colored red in the Figure 23(b), point to a new segment S11. The recovery process will traverse the buddy tree and visit directory[8]. Since the stride of S2 is 4, the recovery process will make sure directory[9], directory[10], and directory[11] have the same local depth and point to the same segment. Since directory[11] points to a different segment, we can detect the inconsistency and fix it by restoring its pointer. If a system crashes after we update directory[10] and directory[11] as shown in Figure 23(c), we can either restore the two buddies or increase the local depth of directory[8] and directory[9].

#### 4.4.6 Concurrency and Consistency Model

Rehashing is particularly challenging when a large number of transactions are concurrently running because rehashing requires all concurrent write threads to wait until rehashing is complete.



---

**Algorithm 4** Directory Recovery
 

---

```

1: while  $i < \text{Directory.Capacity}$  do
2:    $\text{Depth}_{Cur} \leftarrow \text{Directory}[i].\text{Depth}_{local}$ 
3:    $\text{Stride} \leftarrow 2^{(\text{Depth}_{global} - \text{Depth}_{Cur})}$ 
4:    $j \leftarrow i + \text{Stride}$  ▷ Buddy Index
5:    $\text{Depth}_{Buddy} \leftarrow \text{Directory}[j].\text{Depth}_{local}$ 
6:   if  $\text{Depth}_{Cur} < \text{Depth}_{Buddy}$  then
7:     for  $k \leftarrow j - 1; i < k; k \leftarrow k - 1$  do
8:        $\text{Directory}[k].\text{Depth}_{local} \leftarrow \text{Depth}_{Cur}$ 
9:     end for
10:  else
11:    if  $\text{Depth}_{Cur} = \text{Depth}_{Buddy}$  then
12:      for  $k \leftarrow j + 1; k < j + \text{Stride}; k \leftarrow k + 1$  do
13:         $\text{Directory}[k] \leftarrow \text{Directory}[j]$ 
14:      end for
15:    else ▷  $\text{Depth}_{Cur} > \text{Depth}_{Buddy}$ ; Shrink
16:      for  $k \leftarrow j + \text{Stride} - 1; j \leq k; k \leftarrow k - 1$  do
17:         $\text{Directory}[k] \leftarrow \text{Directory}[j + \text{Stride} - 1]$ 
18:      end for
19:    end if
20:  end if
21:   $i \leftarrow i + 2^{(\text{Depth}_{global} - (\text{Depth}_{Cur} - 1))}$ 
22: end while

```

---

To manage concurrent accesses in a thread-safe way in CCEH, we adapt and make minor modifications to the two level locking scheme proposed by Ellis [86], which is known to show reasonable performance for extendible hashing [87]. For buckets, we protect them using a reader/writer lock. For segments, we have two options. One option is that we protect each segment using a reader/writer lock as with buckets. The other option is the lock-free access to segments.

Let us first describe the default reader/writer lock option. Although making use of a reader/writer lock for each segment access is expensive, this is necessary because of the in-place lazy deletion algorithm that we described in Section 4.4.2. Suppose a read transaction T1 visits a segment but goes to sleep before reading a record in the segment. If we do not protect the segment using a reader/writer lock, another write transaction T2 can split the segment and migrate the record to a new segment. Then, another transaction accesses the split segment and overwrites the record that the sleeping transaction is to read. Later, transaction T1 will not find the record although the record exists in the new buddy segment.

The other option is lock-free access. Although lock-free search cannot enforce the ordering of transactions, which makes queries vulnerable to *phantom and dirty reads* problems [51], it is



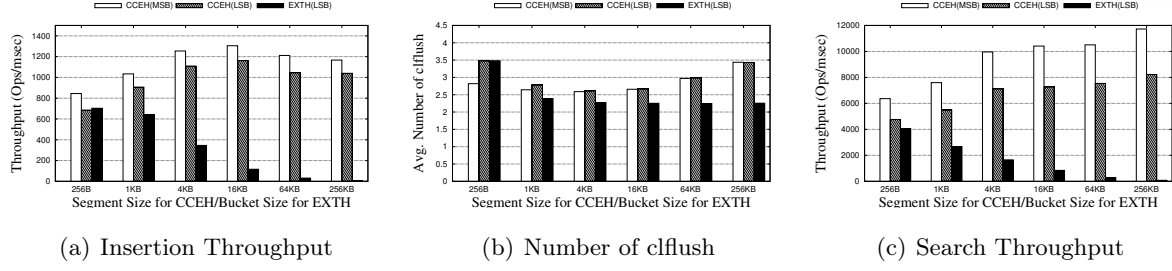


Figure 24: *Throughput with Varying Segment/Bucket Size*

useful for certain types of queries, such as OLAP queries, that do not require a strong consistency model because lock-free search helps reduce query latency.

To enable lock-free search in CCEH, we cannot use the lazy deletion and in-place updates. Instead, we can copy-on-write (CoW) split segments. With CoW split, we do not overwrite any existing record in the split segment. Therefore, a lock-free query accesses the old split segment until we replace the pointer in the directory with a new segment. Unless we immediately deallocate the split segment, the read query can find the correct key-value records even after the split segment is replaced by two new segments. To deallocate the split segment in a thread-safe way, we keep count of how many read transactions are referencing the split segment. If the reference count becomes zero, we ask the persistent heap memory manager to deallocate the segment. As such, a write transaction can split a segment even while it is being accessed by read transactions.

We note that the default CCEH with lazy deletion has a much smaller overhead for segment split than the CCEH with CoW split, which we denote as CCEH(C), because it reuses the original segment so that it can allocate and copy only half the amount required for CCEH(C). If a system failure occurs during a segment split, the recovery cost for lazy deletion is also only half of that of CCEH(C). On the other hand, CCEH(C) that enables lock-free search at the cost of weak consistency guarantee and higher split overhead shows faster and more scalable search performance, as we will show in Section 4.4.7. Another benefit of CCEH(C) is that its probing cost for search operations is smaller than that of CCEH with lazy deletion because all the invalid keys are overwritten as NULL.

For more scalable systems, lock-free extendible hashing has been studied by Shalev et al. [88]. However, such lock-free extendible hashing manages each key-value record as a *split-ordered* list, which fails to leverage memory level parallelism and suffers from a large number of cacheline accesses.

To minimize the impact of rehashing and reduce the tail latency, numerous hash table implementations including Java Concurrent Package and Intel Thread Building Block partition the hash table into small regions and use an exclusive lock for each region [89, 90, 91, 92], hence avoiding full-table rehashing. Such region-based rehashing is similar to our CCEH in the sense that CCEH rehashes only one segment at a time. However, we note that the existing region-

based concurrent hash table implementations are not designed to guarantee failure-atomicity for PM. Furthermore, their concurrent hash tables use separate chaining hash tables, not dynamic hash tables [89, 90, 91, 92].

#### 4.4.7 Experiments

We run experiments on a workstation that has four Intel Xeon Haswell-EX E7-4809 v3 processors (8 cores, 2.0GHz, 8×32KB instruction cache, 8×32KB data cache, 8×256KB L2 cache, and 20MB L3 cache) and 64GB of DDR3 DRAM. Since byte-addressable persistent main memory is not commercially available yet, we emulate persistent memory using *Quartz*, a DRAM-based PM latency emulator [83, 82]. To emulate write latency, we inject stall cycles after each `clflush` instructions, as was done in previous studies [84, 39, 27, 33, 85].

A major reason to use dynamic hashing over static hashing is to dynamically expand or shrink hash table sizes. Therefore, we set the initial hash table sizes such that they can store only a maximum of 2048 records. For all experiments, we insert 16 million random keys, whose keys and values are of 8 bytes. Although we do not show experimental results for non-uniformly distributed keys such as skewed distributions due to the page limit, the results are similar because well designed hash functions convert a non-uniform distribution into one that is close to uniform [93].

##### 4.4.7.1 Quantification of CCEH Design

In the first set of experiments, we quantify the performance effect of each design of CCEH. Figure 24 shows the insertion throughput and the number of cacheline flushes when we insert 16 million records into variants of the extendible hash table, while increasing the size of the memory blocks pointed by directory entries, i.e., the segment in CCEH and the hash bucket in extendible hashing. We fix the size of the bucket in CCEH to a single cacheline, but employ linear probing and bound the probing distance to four cachelines to maximize memory level parallelism.

CCEH(MSB) and CCEH(LSB) show the performance of CCEH when using MSB and LSB bits, respectively, as the segment index and LSB and MSB bits, respectively, as the bucket index. EXTH(LSB) shows the performance of legacy extendible hashing that uses LSB as the bucket index, which is the popular practice.

When the bucket size is 256 bytes, each insertion into EXTH(LSB) calls `clflush` instructions about 3.5 times on average. Considering an insertion without a collision requires only a single `clflush` to store a record in a bucket, 2.5 cacheline flushes are the amortized cost of bucket splits and directory doubling. Note that CCEH(LSB) and EXTH(LSB) are the same hash tables when a segment can hold a single bucket. Therefore, their throughputs and number of cacheline accesses are similar when the segment size of CCEH(LSB) and the bucket size of EXTH(LSB) are 256 bytes.

As we increase the bucket size, EXTH(LSB) splits buckets less frequently, decreasing the

number of `clflush` down to 2.3. However, despite the fewer number of `clflush` calls, the insertion throughput of EXTH(LSB) decreases sharply as we increase the bucket size. This is because EXTH(LSB) reads a larger number of cachelines when finding a record as the bucket size increases.

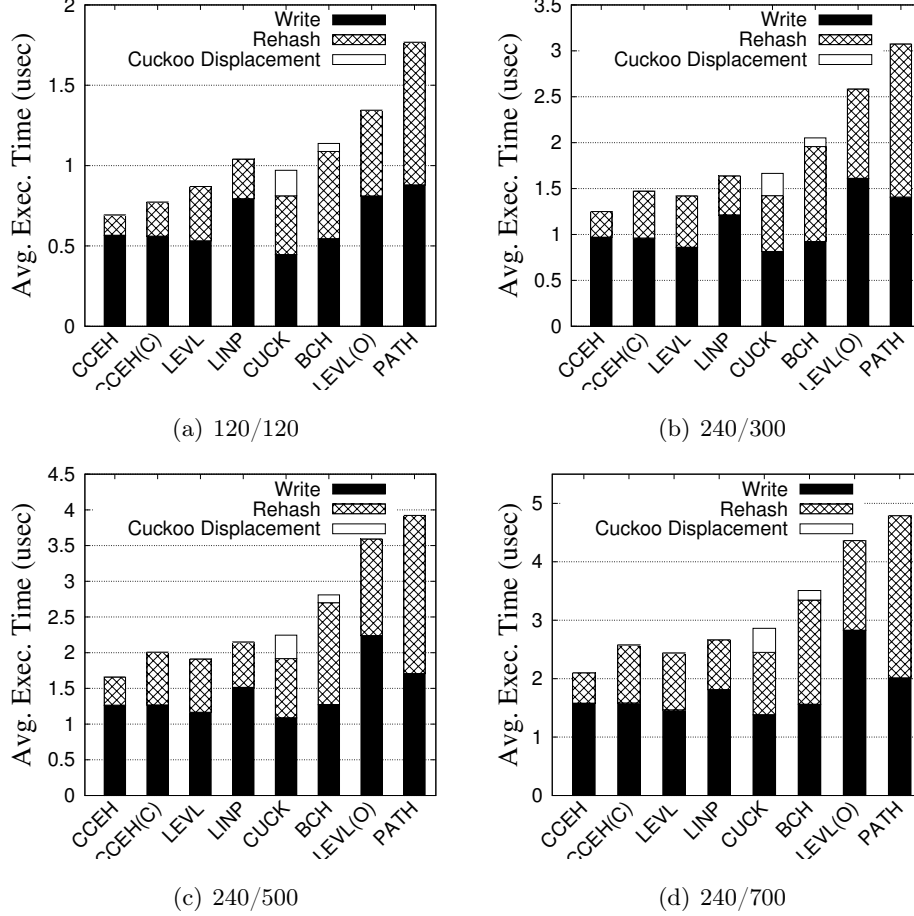


Figure 25: *Breakdown of Time Spent for Insertion While Varying R/W latency of PM*

In contrast, as we increase the segment size up to 16KB, the insertion throughput of CCEH(MSB) and CCEH(LSB) increase because a segment split occurs less frequently while the number of cachelines to read, i.e., LLC misses, is not affected by the large segment size. However, if the segment size is larger than 16KB, the segment split results in a large number of cacheline flushes, which starts degrading the insertion throughput.

Figure 24(b) shows CCEH(MSB) and CCEH(LSB) calling a larger number of `clflush` than EXTH(LSB) when a segment or bucket splits. This is because CCEH(MSB) and CCEH(LSB) store records in a sparse manner according to the bucket index whereas EXTH(LSB) sequentially stores rehashed records without fragmented free spaces. Thus, the number of updated cachelines written by EXTH(LSB) is only about half of CCEH(LSB) and CCEH(MSB). From the experiments, we observe the reasonable segment size is in the range of 4KB to 16KB.

When the segment size is small, the amortized cost of segment splits in CCEH(MSB) is up

to 29% smaller than that of CCEH(LSB) because CCEH(MSB) updates adjacent directory entries, minimizing the number of `clflush` instructions. However, CCEH(LSB) accesses scattered cache-lines and fails to leverage memory level parallelism, which results in about 10% higher insertion time on average.

#### 4.4.7.2 Comparative Performance

For the rest of the experiments, we use a single byte as the bucket index such that the bucket size is 16 Kbytes, and we do not show the performance of CCEH(LSB) since CCEH(MSB) consistently outperforms CCEH(LSB). We compare the performance of CCEH against legacy hash table with linear probing (LINP), cuckoo hashing [56] (CUCK), bucketized cuckoo hashing (BCH) [94], path hashing [54] (PATH), and level hashing [53] (LEVL and LEVL(0)).<sup>1</sup>

For path hashing, we set the reserved level to 8, which guarantees 92% maximum load factor as suggested by the authors [54]. For BCH, we set the bucket size to a cacheline where we can store four key-value records of  $\langle word, word \rangle$  type. We choose a small bucket size for BCH because we observed BCH suffers from more cacheline accesses and shows worse performance as we increase the bucket size. Cuckoo hashing and BCH perform full-table rehashing when they fail to displace a collided record 16 times, which shows the fastest insertion performance on our testbed machine. Linear probing rehashes when the load factor reaches 95%.

In the experiments shown in Figure 25, as the latency for reads and writes of PM are changed, we insert 16 million records in batches and breakdown the insertion time into (1) the bucket search and write time (denoted as **Write**), (2) the rehashing time (denoted as **Rehash**), and (3) the time to displace existing records to another bucket, which is necessary for cuckoo hashing (denoted as **Cuckoo Displacement**).

CCEH shows the fastest average insertion time throughout all read/write latencies. Even if we disable the lazy deletion but perform copy-on-write for segment splits, denoted as CCEH(C), CCEH(C) outperforms LEVL when PM read/write latencies are no different from DRAM latencies. Note that the **Rehash** overhead of CCEH(C) is twice higher than that of CCEH that reuses the split segment via lazy deletion. However, as the write latency of PM increases, CCEH(C) is slightly outperformed by LEVL because of frequent memory allocations and expensive copy-on-write operations. The total allocated memory space for both CCEH(C) and LEVL are similar whereas the total allocated memory space for CCEH is about half since it benefits from reusing split

---

<sup>1</sup>We downloaded the author's level hashing, path hashing, and BCH implementations from <https://github.com/Level-Hashing/level-hashing>. However, we found these implementations from the authors are much slower than our extendible hashing, even slower than the simplest linear probing because they do not align buckets to cachelines. For fair comparisons, we made minor changes to their implementations such that buckets are aligned to cachelines. For reference purpose, we show the performance of original level hash implementation from the authors, denoted as LEVL(0) and compare it with our modified level hash implementation LEVL. Our modified level hashing (LEVL), BCH (BCH), and path hashing (PATH) codes are available at <http://github.com/ccehtable/CCEH>. Our own implementations of CCEH, linear probing (LINP), and cuckoo hashing (CUCK) are also available there.

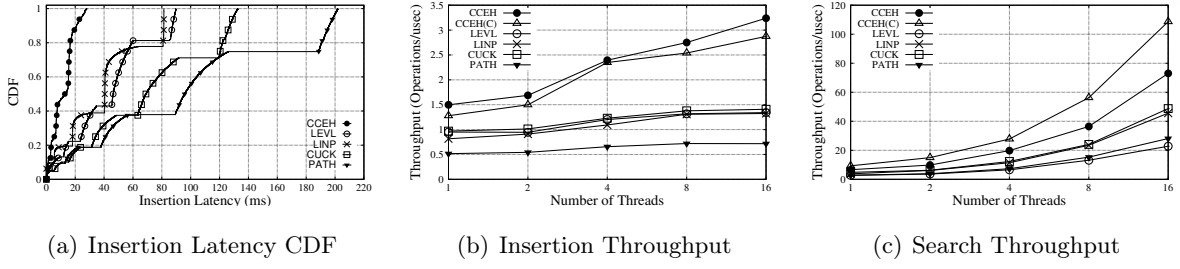


Figure 26: *Performance of concurrent execution: latency CDF and insertion/search throughput*

segments.

Interestingly, the rehashing overhead of LEVL is even higher than that of LINP, which is just a single array that employs linear probing for hash collisions. Although LINP suffers from a large number of cacheline accesses due to the open addressing, its rehashing overhead is not more significant than other static hashing schemes that perform full-table rehashing including LEVL. As we discussed in Section II, the bottom level hash table in LEVL often fails to accommodate a collided record resulting in another rehash. As such, LEVL performs rehashing for 1/3 records, which is soon followed by another rehashing for the rest. Therefore, the average rehashing cost of LEVL is similar to other hashing schemes.

CUCK also uses a single array as in LINP but performs the cuckoo displacements using two hash functions. While a hash function in CUCK determines the exact location of a record, hash functions in BCH compute the location of a bucket that can store multiple records. Although the average insertion times of CUCK and BCH are similar when PM latency is equal to DRAM latency, CUCK displaces a larger number of records and shows higher Cuckoo Displacement overhead than BCH. That is, BCH can accommodate collided records in its large buckets instead of displacing it to another bucket. However, BCH has higher rehashing overhead than CUCK because the BCH implementation calls `clflush` to rehash each record but our CUCK implementation calls a batch `clflush` in a lazy manner to synchronize all dirty cachelines after rehashing is done. Such lazy synchronization is known to decouple the volatile memory order from the persist order [27] and help improving the performance.

PATH hashing shows the worst performance throughout all our experiments mainly because its lookup cost is not constant, but  $O(\log_2 N)$ .

#### 4.4.7.3 Concurrency and Latency

Full-table rehashing is particularly challenging when multiple queries are concurrently accessing a hash table because full-table rehashing requires exclusive access to the entire hash table, which blocks subsequent queries and increases the response time. Therefore, we measure the latency of concurrent insertion queries including the waiting time, whose CDF is shown in Figure 26(a). For the workload, we generated query inter-arrival patterns using Poisson distribution where the  $\lambda$  rate is set to the batch processing throughput of LINP.

While the average batch insertion times differ by only up to 130%, the maximum latency of **PATH** is up to  $8\times$  higher than that of **CCEH** (25 msec vs. 200 msec), as shown in Figure 26(a). This is because full-table rehashing blocks a large number of concurrent queries and significantly increases their waiting time. The length of each flat region in the CDF graph represents how long each full-table rehashing takes. **PATH** takes the longest time for rehashing whereas **LEVL**, **LINP**, and **CUCK** spend similar amounts of time on rehashing. In contrast, we do not find any flat region in the graph for **CCEH**. Compared to **LEVL**, the maximum latency of **CCEH** is reduced by over two-thirds.

For the experimental results shown in Figure 26(b) and (c), we evaluate the performance of the multi-threaded versions of the hashing schemes. Each thread inserts  $16/k$  million records in batches where  $k$  is the number of threads. Overall, as we run a larger number of insertion threads, the insertion throughputs of all hashing schemes improve slightly but not linearly due to lock contention.

Individually, **CCEH** shows slightly higher insertion throughput than **CCEH(C)** because of smaller split overhead. **LEVL**, **LINP**, **CUCK**, and **PATH** use a fine-grained reader/writer lock for each sub-array that contains 256 records, which is even smaller than the segment size of **CCEH**, but they fail to scale because of the rehashing overhead. We note that these static hash tables must obtain exclusive locks for all the fine-grained sub-arrays to perform rehashing. Otherwise, queries will access a stale hash table and return inconsistent records.

In terms of search throughput, **CCEH(C)** outperforms **CCEH** as copy-on-write lock-free search. Since the read transactions of **CCEH(C)** are non-blocking, the search throughput of **CCEH(C)** is 1.46x, 2.2x, and 4.6x higher than that of **CCEH**, **CUCK**, and **LEVL**, respectively. Interestingly, **LEVL**, our modified version of **LEVL(0)**, shows the worst search throughput not only in these experiments, but also in batch search experiments as well, which we do not show due to the page limit. We note that **LEVL(0)** shows even worse search performance than **LEVL**. Our analysis of the level hashing implementation found in github shows that level hashing is using the cuckoo displacement, which accesses discontinuous cachelines multiple times, fails to leverage memory level parallelism, and increases the LLC misses. In addition, level hashing uses a small bucket and performs linear probing inside the bucket, which also increases the number of cacheline accesses and hurts the search performance even more. As a result, **LEVL** shows the worst search throughput. In these experiments, all queries lookup keys that exist in the hash table. However, if queries are made to keys that do not exist in the hash table, making use of long probing, cuckoo displacement, and stash hurts search performance even more. Although the results are not presented in the interest of space, we find that our **CCEH** significantly outperforms other hashing schemes even for non-existent key lookups.



## V Parallel Tree Traversal for Nearest Neighbor Query on the GPU

The nearest neighbor search, also known as proximity search or similarity search is a fundamental problem that finds the closest point to a given query point in multi-dimensional space. The problem of this nearest neighbor search is that the brute-force algorithm usually outperforms the indexing as the dimension increases. This is due to the curse of dimensionality, the exponential growth of hyper-volume as a function of the number of dimensions. As GPU has been widely adopted as a cost-effective solution in various computing domains, we studied several multi-dimensional indexes on the accelerator. One of the challenges to use such data structures is the small shared memory on the GPU. The small size of the shared memory in modern GPUs does not allow storage of more than one tree node in the run-time stack, and the traditional recursive tree traversal algorithms fail due to the stack overflow. Therefore, the tree traversal on the GPU should avoid recursive search algorithms while efficiently using the computation resources on it.

### 5.1 Parallel Scan and Backtrack for kNN Query

In this section, we propose a novel tree traversal algorithm, *Parallel Scan and Backtrack (PSB)*, for kNN search on the GPU. The PSB algorithm does not use a runtime stack in shared memory since the main purpose of shared memory on the GPU is to coordinate concurrent threads and the shared memory is better reserved for application specific purpose, such as, the k-nearest points. Instead, PSB requires each tree node to have a parent link pointer but PSB makes its best efforts to avoid revisiting already visited parent nodes.

The PSB algorithm is shown in Algorithm 5. For a given query point  $Q$  and bounding spheres of child nodes, we compute the minimum distances (MINDIST) and maximum distances (MAXDIST) between the query point and the closest faces of child bounding spheres. If the minimum MAXDIST (MINMAXDIST) is smaller than the current pruning distance, we set the pruning distance to MINMAXDIST. As in the classic branch-and-bound kNN query processing algorithm, the PSB algorithm visits the child node whose MINDIST is smallest until it reaches a leaf node. In the leaf node, the PSB algorithm updates its pruning distance, and restarts the tree traversal from the root node with the small pruning distance. In the second traversal, unlike the classic branch-and-bound kNN search algorithm, the PSB search algorithm visits the leftmost child node in the tree node within the pruning distance. The sub-trees on the left side of the chosen child node are not within the pruning distance, so they can be pruned out without hurting the correctness of the algorithm.

Once the search path reaches the leftmost leaf node within the pruning distance, the PSB algorithm updates its pruning distance and kNN points if it finds closer data points to the query. While the classic branch-and-bound kNN search algorithm goes back to the parent node after it visits a leaf node, the PSB algorithm starts scanning sibling leaf nodes. This is because the

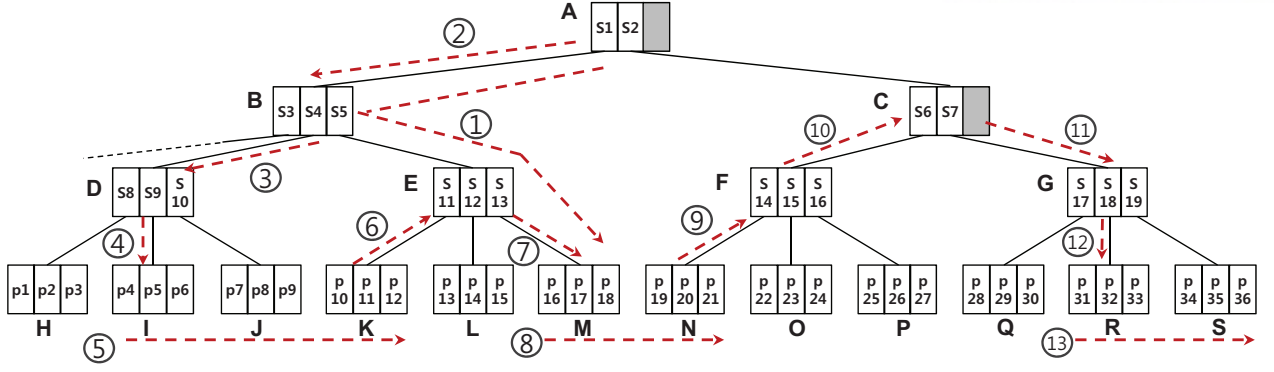


Figure 27: *Massively Parallel Scanning and Backtracking*: In the root node  $A$ , the pruning distance is initially infinite. In step ①, we search a leaf node which is closest to the query point and update the pruning distance while computing the MAXDIST of child nodes. In the second tree traversal ②, regardless of whether  $B$  or  $C$  is closer to the query point, we fetch the leftmost child node  $B$  from global memory if both  $B$  and  $C$  are within pruning distance. In node  $B$ , we check which child nodes are within the pruning distance. In the example, suppose  $D$  and  $E$  are within the pruning distance. Then ③ we fetch the left child node  $D$ . Suppose  $H$  is not within the pruning distance. Then node  $H$  will be pruned out, and ④ we visit node  $I$  and kNN points will be updated. After processing node  $I$ , ⑤ we fetch its sibling nodes  $J$  and  $K$ . If node  $K$  does not update kNN points or pruning distance, ⑥ we fetch  $K$ 's parent node  $E$  from global memory and prune out child nodes ( $L$  in the example) which are farther than the pruning distance. If  $M$ 's MINDIST is smaller than the pruning distance, ⑦ we fetch  $M$  and ⑧ keep scanning its sibling nodes. If node  $N$  does not update kNN points, ⑨ we move to its parent node  $F$ , which does not have any child nodes within the pruning distance. Thus ⑩ we move one level up to node  $C$ . If node  $C$  does not have any child nodes within the pruning distance, we will move to the root node and finish the search. Otherwise, as in the example, we visit the leftmost leaf node  $G$  as it is within the pruning distance. We keep this traversal and visit  $G$ ,  $R$ , and  $S$ .

sibling nodes have a high chance of having points spatially close to the query point because the leaf nodes are likely to be clustered in the problem space. If we encounter a leaf node that does not update the kNN points while scanning the sibling leaf nodes, there is no point in scanning further. Thus the PSB algorithm stops scanning sibling leaf nodes and follows its parent link so that it fetches the parent node of the last visited leaf node. Due to the leaf node scanning, the parent node is less likely to have been already visited. If the parent node has unvisited child nodes within the pruning distance, again we choose and visit the leftmost node among the unvisited child nodes. Otherwise we repeatedly move to the upper-level parent node following parent links. The PSB tree traversal algorithm visits leaf nodes in a sequential fashion, and a large number of GPU cores helps process them in a massively parallel fashion. Figure 2 shows an example of the PSB tree traversal algorithm.



## 5.2 Bottom-up Construction of SS-tree

The classic SS-trees construct hierarchical tree structures in a top-down fashion. That is, when a point is inserted, the SS-tree insertion algorithm determines which subtree's centroid is closest to the point and inserts it into that sub-tree. When a node overflows, the split algorithm calculates the coordinate variance from the centroid in each dimension. Based on the variance, it chooses the dimension with the highest variance and splits the overflowed node along that dimension. This split algorithm is known to divide multi-dimensional points into isotropic bounding spheres and greatly reduces the sizes of bounding spheres. As a heuristic optimization, SS-tree employs *forced reinsertion*, as in R\*-tree, so that it dynamically reorganizes the tree structure and reduces the amount of overlap between bounding spheres.

Such a top-down and sequential construction algorithm requires serialization of insert operations and excessive locking. If a data point is inserted online, top-down insertion will do the work, but when we need to create an index in batches, bottom-up construction can create an index an order of magnitude faster, as in Packed R-tree [95]. Moreover, the bottom-up construction can take advantage of high level parallelism on the GPU.

### 5.2.1 Bottom-up Construction using Hilbert Curve

The Hilbert curve is a space filling curve that is widely used in many computing domains because it is known to preserve good spatial locality [96]. Using the Hilbert space filling curve, we can determine the ordering of multi-dimensional points. With the sorted ordering, we cluster nearby points, enclose the nearby points in a small bounding sphere, and store them in a leaf node. Although the Hilbert space filling curve can assign distant index values to spatially close data points, it guarantees that it does not assign similar index values to distant data points. This is the desirable property that helps generate tight bounding spheres in leaf nodes.

The Hilbert index values of multi-dimensional points can be concurrently calculated via task parallelism. Moreover, the Hilbert index values can be efficiently sorted in parallel on the GPU. Parallel sorting on the GPU has been extensively studied in the past decade. In our implementation, we employ the parallel radix sort available in the *Thrust* [97] CUDA library.

### 5.2.2 Bottom-up Construction using K-Means Clustering

In addition to the parallel construction using the Hilbert curve, we develop an alternative parallel SS-tree construction algorithm using k-means clustering. K-means clustering partitions a given set of multi-dimensional points into  $k$  clusters, and we store each cluster in a SS-tree leaf node. A challenge in using the k-means clustering method is to determine the number of  $k$  since it is not known a priori. In general, increasing  $k$  results in reducing the potential errors. As a rule of thumb, we can set  $k$  to the number of leaf nodes ( $NumberOfPoints/CapacityOfLeafNode$ ). However, the k-means clustering algorithm does not guarantee that all clusters have an equal number of data points. Also, as  $k$  increases, the time to compute the clusters increases expo-

nentially because its time complexity is  $O(n^{dk+1} \log n)$ , where  $n$  is the number of points and  $d$  is the number of dimensions. With a small  $k$ , data points in a single cluster can be distributed across multiple leaf nodes. In our implementation, we set  $k$  to  $\sqrt{n/2}$ , where  $n$  is the number of points, as proposed by Mardia et al. [98].

### 5.2.3 Bottom-up Construction of Hierarchical Minimum Enclosing Bounding Spheres

Once we classify data points and store them in leaf nodes, we repeat recursive construction of minimum enclosing bounding spheres for internal tree nodes until we create a bounding sphere for the root node. The smallest enclosing circle in 2D space can be found in  $O(n)$  time. However, its time complexity increases sharply as the dimension increases, i.e., the complexity of Megiddo's linear programming algorithm is  $O((d+1)(d+1)!n)$  [99]. For high dimensional points, a large number of approximation algorithms, including Ritter's algorithm [4], have been proposed. The approximation algorithms generate bounding spheres fast but the bounding spheres are slightly larger than optimum. That is, Ritter's algorithm is known to generate 5~20% larger bounding spheres. Although the tree construction is only a one time job, running an  $O((d+1)(d+1)!n)$  algorithm is not practical in practice. Thus, we employ Ritter's approximation algorithm which is easy to parallelize as we describe in Algorithm 6. To the best of our knowledge, not much but just a little research [100, 101, 102] has been conducted to parallelize the construction of minimum enclosing bounding spheres, especially in high dimensions. We believe this is the first work that develops a parallel version of Ritter's algorithm. The parallel Ritter's algorithm is described in Algorithm 6.

In our parallel Ritter's algorithm, we choose a random point  $p$  from a set of points  $S$ , and compute the distances between  $p$  and the rest of the points in parallel. Once the distances are computed, we perform parallel reduction to choose the point  $q$  that has the largest distance from  $p$ . From  $q$ , we compute the distances between  $q$  and the rest of the points in parallel, and choose the farthest point  $r$ . Once we find  $q$  and  $r$ , we create an initial sphere using  $\overline{qr}$  as its diameter. Next, we check if all points in  $S$  are included in the sphere by simply checking the distances between the centroid ( $C1$ ) and each point in parallel. If a point  $s$  is outside of the sphere, we draw a line between  $C1$  and  $s$  and extend the sphere toward  $s$  just enough to include  $s$ . We repeat this process until all the points are included in the sphere.

Note that our bottom-up parallel construction algorithm enforces 100 % node utilization of leaf nodes even if we can significantly reduce the volume by storing some points in a sibling tree node. However, as the node utilization of the bottom-up constructed SS-tree is higher than that of the classic SS-tree, the number of tree nodes is smaller than the classic SS-tree, which results in a shorter search path.

#### 5.2.4 Performance Evaluation of Bottom-up Constructed SS-tree

In the experiments shown in Figure 28, we compare the search performance of SS-trees constructed in a bottom-up fashion using the Hilbert curve and k-means clustering. We run the experiments using NVIDIA K40 GPUs for the bottom-up SS-tree implementations, and we set the degree of a SS-tree node to 128 so that each processing unit in a shared multiprocessor processes four branches, i.e., a total of 32 branches are processed in parallel. We also compare the search performance of bottom-up constructed SS-tree on the GPU with that of top-down constructed SR-tree on a CPU. The SR-tree is an improved version of the SS-tree [65]. For SR-tree, we run the experiments with an Intel Xeon E5-2690 v2 CPU, and set the size of a tree node to a disk page size - 8 Kbytes. However, the search time comparison on two different architectures is like a comparison of apples and oranges. Therefore, we compare the search performance in terms of the number of accessed bytes as well.

We synthetically generate 100 sets of multi-dimensional points in normal distributions with various average points and standard deviations. Each distribution consists of 10,000 data points. Therefore, the total number of points in the dataset is one million. We describe how the distribution of a dataset affects the performance of indexing in more detail in section 5.3. For the k-means clustering algorithm, we vary the  $k$  from 400 to 10,000 at the leaf node level. For the clustering of internal tree nodes, we decrease  $k$  by a factor of 1/100 since the number of internal tree nodes is much less than that of leaf nodes.

As for the tree traversal algorithm, we use the classic branch-and-bound kNN query processing algorithm [64] for all indexes because the goal of the experiments is to evaluate how the construction algorithm affects the tree structures. However, because the SS-tree on the GPU does not allow backtracking, we let the SS-tree on the GPU use auxiliary *parent links* so that it can backtrack to the parent nodes.

As shown in Figure 28, SS-trees that we construct via k-means clustering algorithms (**SS-tree (k-means)**) consistently outperform SS-trees that we construct via Hilbert curve clustering (**SS-tree (Hilbert)**). In four dimensions, the bottom-up **SS-tree (Hilbert)** accesses about 16 times more tree nodes than the bottom-up **SS-tree (k-means)**, which results in 7.1 times slower average query response time. Compared to SR-tree, SS-trees constructed via the Hilbert curve and k-means clustering access about a 4~16 times larger number of tree nodes. This is because Ritter's algorithm does not find the optimal minimum bounding spheres, and it is also because SS-trees visit the same tree nodes multiple times using parent links. Although SS-trees on the GPU access a much larger number of tree nodes, they outperform SR-trees in terms of search response time due to massive parallelism.

As for the k-means clustering,  $k = 400$  shows the best performance but as  $k$  is larger or smaller than that, the performance slightly degrades. Since the distribution of the datasets is not known a priori most of the time, it is a hard problem to choose the optimal  $k$  as we discussed. However, although we choose  $k$  far from the optimal value, the SS-tree with k-means clustering

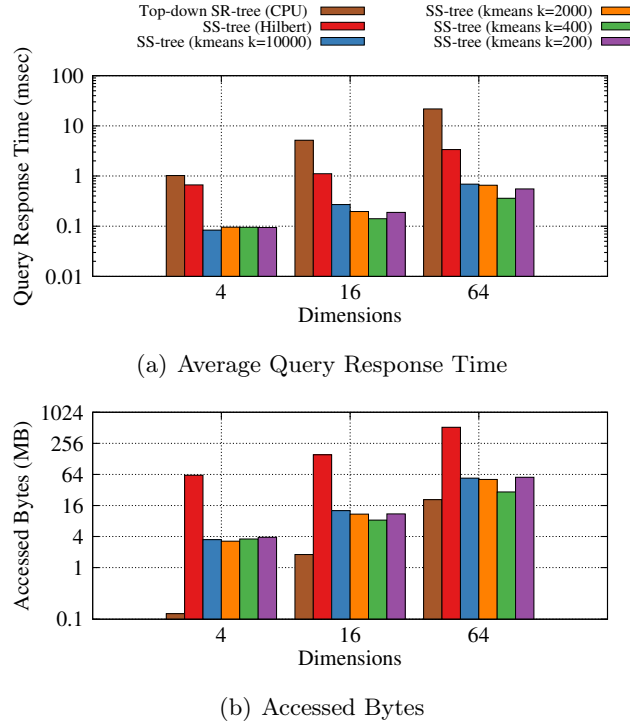


Figure 28: *Bottom-up Constructed SS-trees vs Top-down Constructed SR-tree (Parent Link Tree Traversal)*

algorithm consistently outperforms the SS-tree that employs the Hilbert curve as shown in the experiments.

## 5.3 Experiments

### 5.3.1 Experimental Setup

In this section, we evaluate and analyze the performance of the PSB (Parallel Scan and Back-track) tree traversal algorithm for the nearest neighbor query processing algorithm on the GPU. We conduct the experiments on a CentOS Linux machine that has dual Intel Xeon E5-2640v2 2.0GHz processors and 64 GB DDR3 memory with an NVIDIA Tesla K40 GPU which has 2880 CUDA cores. We use CUDA 6.5 for all the experiments. In our implementation of SS-trees, we store the bounding spheres of child nodes as the structure of array (SOA) instead of the array of structure so that memory coalescing can be naturally employed. We compiled the codes with default optimization options using nvcc 6.5.12 and gcc 4.4.7.

The datasets of our primary concern are the non-uniform clustered datasets since the nearest neighbor search in high dimensional uniform distribution is not even meaningful; that is, Beyer et al. proved in [103] that the distance to the farthest neighbor and the distance to the nearest neighbor converge as the dimension increases to infinity. We evaluate the performance of the proposed algorithm with various synthetic multi-dimensional datasets. We vary the number of dimensions from 2 to 64, and we also vary the distribution of the points by changing the number

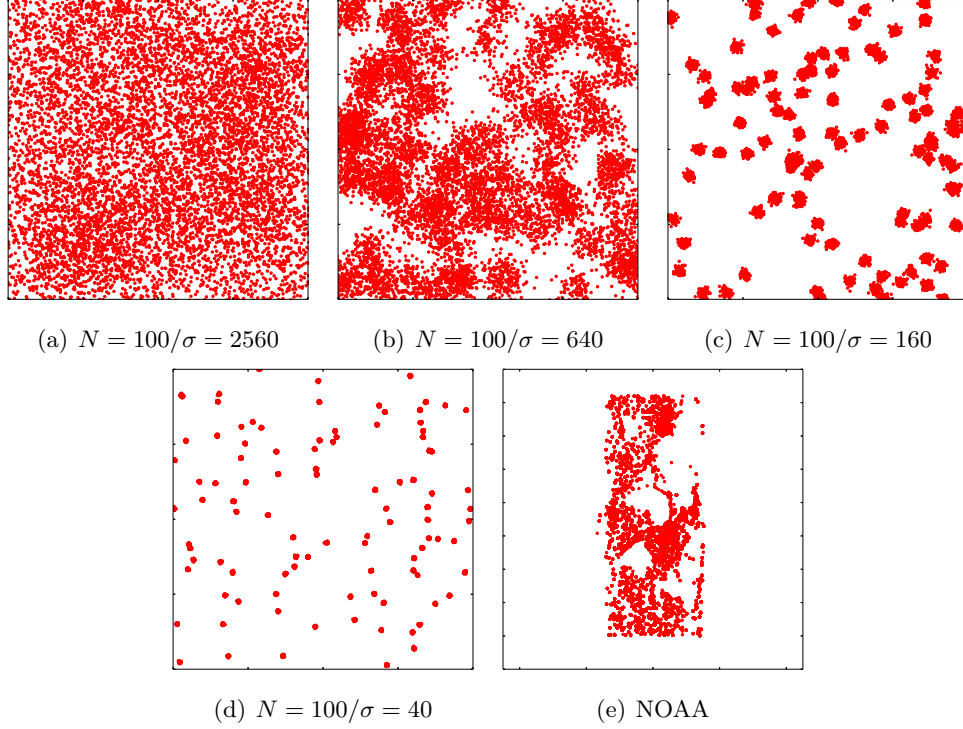


Figure 29: *Distribution of Datasets Projected to the First Two Dimensions ( $N$ : number of clusters,  $\sigma$ : standard deviation)*

of normal distribution clusters and also by changing the variance of each cluster. We carefully adjusted the number of clusters and the variance of the synthetic datasets so that the probability density function of the distances between two arbitrary points is not concentrated.

As we increase the number of clusters and their variances, the distribution of data points becomes similar to the uniform distribution. Figure 29 shows the distributions of the synthetically generated datasets while varying the standard deviation of each cluster's normal distribution.

In addition to the synthetic datasets, we also evaluate the indexing performance using real datasets - Integrated Surface Database (ISD) point datasets available at NOAA National Climatic Data Center. The NOAA datasets consist of numerous sensor values such as wind speed and direction, temperature, pressure, precipitation, etc, collected by over 20,000 geographically distributed stations. The sensor values are tagged with time and two-dimensional coordinates (latitude and longitude).

As for the performance metrics, we measure the average kNN query response time, which is the time for the GPU kernel function to return the search results back to the CPU host for a single query. We also measure the warp efficiency of the GPU and the number of accessed memory bytes.

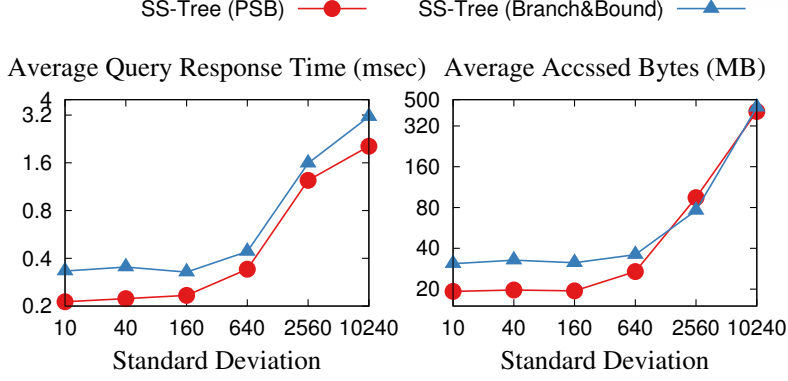


Figure 30: *Query Processing Performance with Varying Input Distribution (100 clusters)*

### 5.3.2 Dataset Distribution

For the experiments shown in Figure 30, we combined 100 normal distributions that have different average points in 64 dimensions. With varying the standard deviations of the distributions, we evaluate the performance of PSB algorithm and the classic branch-and-bound algorithm using bottom-up constructed SS-trees. We submit 240 kNN queries, and each query selects 32 nearest neighbor points from a million data points for the rest of the experiments unless explicitly specified.

In the experiments, we observe the data distribution significantly affects the efficiency of indexing schemes; when the standard deviation is 10240, the query response time is about 8 times higher than when the standard deviation is 40. As the standard deviation increases, the distribution becomes similar to the uniform distribution. With high standard deviation, both the branch-and-bound algorithm and the PSB algorithm visit almost all leaf nodes due to the curse of dimensionality problem. When the standard deviation is higher than 640 in the experiments, the classic branch-and-bound algorithm and the PSB algorithm access a similar number of tree nodes. However, in terms of the query response time, the PSB algorithm consistently outperforms the branch-and-bound algorithm because the PSB algorithm benefits from fast linear scanning.

### 5.3.3 Data Parallel $n$ -ary SS-Tree vs Task Parallel Binary Kd-Tree

In the experiments shown in Figure 31 we vary the number of child nodes (*degree*) and measure the average query execution time, global memory access, and the warp execution efficiency. For the experiments, we index the 64 dimensional synthetic dataset that combines 100 normal distributions, and we set the standard deviations of the distributions to 160. To validate the proposition of this work, we compare the performance of data parallel SS-trees with the PSB algorithm against task parallel binary kd-trees optimized for GPU [104].

When the degree is 32, we let 32 GPU threads concurrently access the same SS-tree node, i.e., the distances between a given query point and 32 bounding spheres of child nodes are calculated

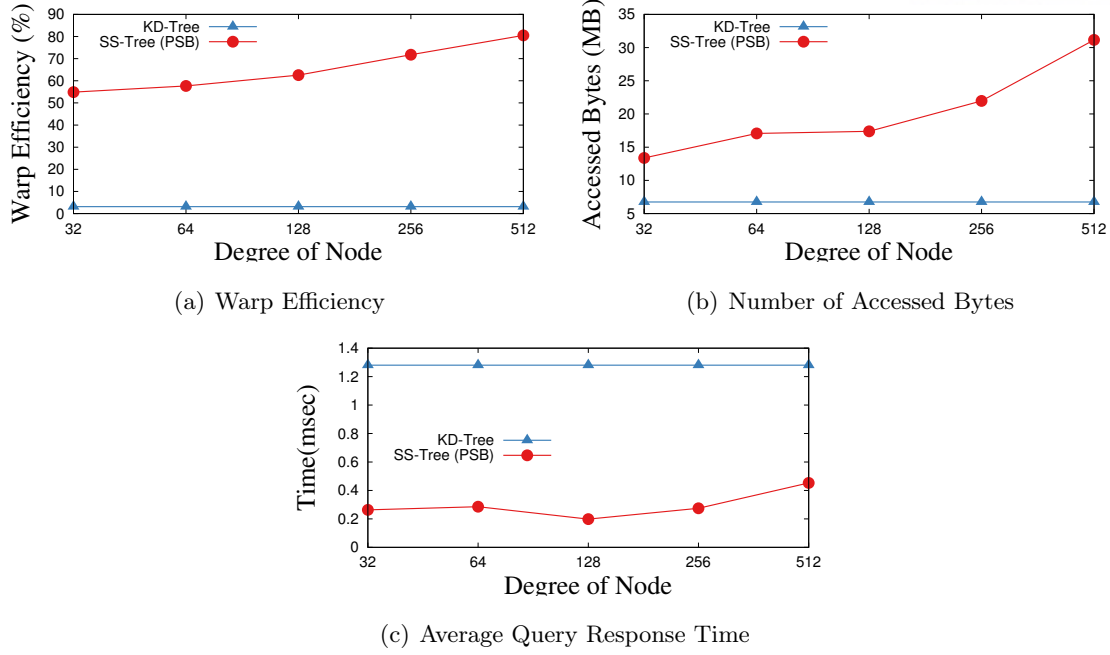


Figure 31: *Query Processing Performance with Varying Number of Fan-outs*

in parallel. The binary kd-tree uses only one core of an SM. Therefore, the warp efficiency of the binary kd-tree is just 3%, but the warp efficiency of the data parallel SS-tree is higher than 50%.

As we increase the degree of a tree node, the SS-tree accesses more global memory due to a larger tree node size. When the degree of the SS-tree node is 4 times larger than the warp size, the query response time slightly degrades because each core has to process more comparisons. But when the degree of the SS-tree node is too small (less than 128), the query response time also slows because the search path length increases.

As we increase the degree of the SS-tree, an SM has more work to do and the idle time of the GPU cores decreases. Although we do not show the query processing throughput results due to space limitation, the data parallel SS-tree shows comparable query processing throughput with the task parallel kd-tree.

#### 5.3.4 Performance in Varying Dimensions

In the experiments shown in Figure 32, we compare the performance of bottom-up constructed SS-tree with the PSB, the branch-and-bound, and the brute-force scanning algorithm on the GPU while varying the dimensions of the data points.

When the datasets are in uniform or Zipf's distribution, it is known that brute-force exhaustive scanning often performs better than indexing structures in high dimensions. However, for the clustered datasets, SS-trees access fewer bytes in global memory and yield faster query response time, as shown in Figure 32. In 64 dimensions, the PSB algorithm performs kNN queries about 4 times faster than brute-force exhaustive scanning, and about 25% faster than



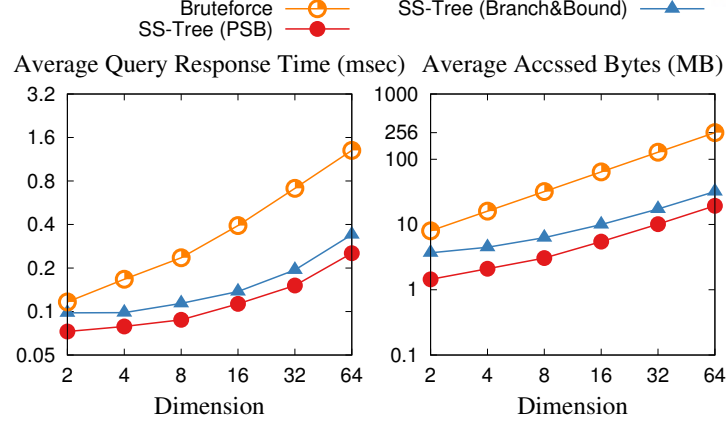


Figure 32: *Performance with Varying Dimensions (Synthetic Datasets (100 clusters))*

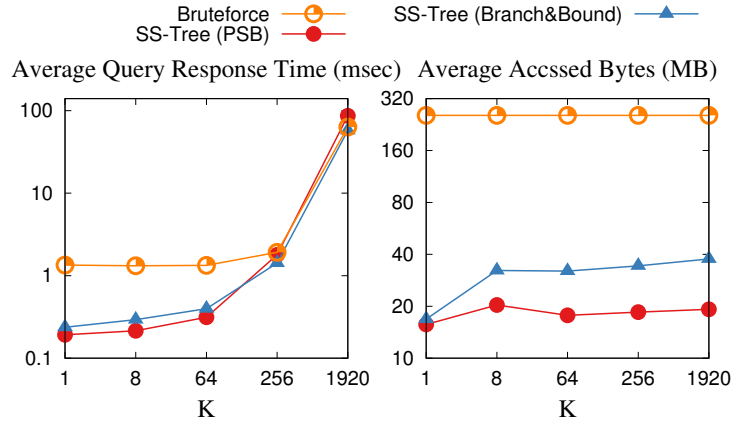


Figure 33: *Query Processing Performance with Varying  $k$*

the classic branch-and-bound algorithm.

### 5.3.5 Performance Effect of K

In the experiments shown in Figure 33, we compare the performance of the PSB, the branch-and-bound algorithm, and the brute-force scanning method while varying the number of nearest neighbor points ( $k$ ). Interestingly, as we increase  $k$ , the query response time increases exponentially although it does not significantly increase the number of accessed tree nodes. This is because we store  $k$  of pruning distances in the shared memory because they must be shared and updated by a block of GPU threads.

As we use more shared memory to store more distances and nearest neighbors, the number of active threads per SM (*GPU occupancy*) decreases. Hence, even the brute-force scanning method suffers from the large  $k$ . In order to maximize the GPU occupancy, the shared memory usage must be limited. As an ad hoc optimization, if a large number of nearest neighbors need to be stored, we can keep only a couple of large pruning distances in the shared memory but the rest of the small pruning distances in global memory because the large pruning distances are more likely to be accessed and updated while the small pruning distances are rarely updated.



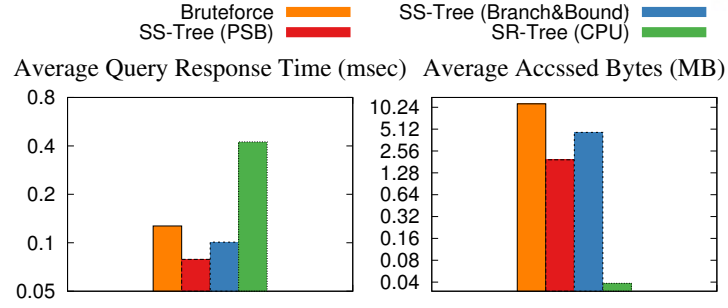


Figure 34: *Query Processing Performance with Real Datasets (NOAA)*

We leave this improvement as our future work.

### 5.3.6 Real Datasets

In the experiments shown in Figure 34, we construct bottom-up constructed SS-trees using real clustered datasets - NOAA. Similar to our synthetic datasets, the PSB algorithm shows superior performance to the branch-and-bound algorithm and the brute-force scanning algorithm. Also, we show the search performance of SR-trees on the GPU. Although the top-down constructed SR-tree accesses a much smaller amount of memory than the bottom-up constructed SS-tree, SR-tree on the CPU shows the worse query response time than the SS-tree on the GPU because of the lack of parallelism.

---

**Algorithm 5** psb algorithm for knn query processing
 

---

**procedure**
*SearchKNN(Point Q, int k)*

```

1: Node n ← root
2: kNNs ← { inf }
3: float pruningDist ← getInitialPruningDistance(Q, ClosestLeaf)
4: int lastleafid ← the sequence id of the rightmost leaf node
5: int visitedleafid ← 0
6: while visitedleafid < lastleafid do // we visit leaf nodes from left to right
7:   while n.level != leaf do
8:     parfor tid ← 1, numchildnodes do
9:       maxdist[tid] ← getmaxdistance(q,n.child[tid])
10:      mindist[tid] ← getmindistance(q,n.child[tid])
11:     end parfor
12:     maxdist[0] ← parreducefindkthminmaxdist(maxdist)
13:     pruningdist ← min(maxdist[0],pruningdist) // parallel reduction to find minmaxdist
14:     for i ← 1, numchildnodes do
15:       if mindist[i] < pruningdist then
16:         n ← n.child[min(i)]
17:         if n.subtreemaxleafid < visitedleafid then // already visited leaf nodes of this subtree
18:           continue
19:         else
20:           break
21:         end if
22:       end if
23:     end for
24:     if i == numchildnode then // no child node is within pruning distance
25:       n ← n.parent // backtrack
26:     end if
27:   end while
28:   while n is a leaf node do
29:     parfor tid ← 1, numthreads do
30:       dist[tid] ← getdistance(q,n.dataptr[tid])
31:     end parfor
32:     update knns with dist[]
33:     update pruningdist with dist[]
34:     visitedleafid ← n.leafid
35:     if there was any change to knns then // scan to the right sibling leaf node
36:       n ← n.rightsibling
37:     else // backtrack to the parent node
38:       n ← n.parent
39:     end if
40:   end while
41: end while
42: return knns

```

---

---

**Algorithm 6** Parallel Ritter's Algorithm
 

---

**procedure**
*ParallelRitter(Node n)*

```

1: float distances[ ]  $\leftarrow$  { inf }
2: parfor  $t \leftarrow 0, n.count$  do
3:   distances[ $t$ ]  $\leftarrow$  Distance( $n.child[0]$ ,  $n.child[t]$ )
4: end parfor
5: distances[0] = parReduceFindMaxDist(distances)
6: int pIdx  $\leftarrow$  Child node index of distances[0] // pIdx is the farthest point from 0
7: parfor  $t \leftarrow 0, n.count$  do
8:   distances[ $t$ ]  $\leftarrow$  Distance( $n.child[pIdx]$ ,  $n.child[t]$ )
9: end parfor
10: distances[0] = parReduceFindMaxDist(distances[])
11: int pIdx2  $\leftarrow$  Child node index of distances[0] // pIdx2 is the farthest point from pIdx
12:  $n.center \leftarrow$  Midpoint of  $n.child[pIdx]$  and  $n.child[pIdx2]$ 
13:  $n.radius \leftarrow$  distances[0]/2
14: bool isUpdated  $\leftarrow$  True
15: while isUpdated = True do
16:   isUpdated  $\leftarrow$  False
17:   parfor  $t \leftarrow 0, n.count$  do
18:     distances[ $t$ ]  $\leftarrow$  Distance from  $n.center$  to  $n.child[t]$ 
19:   end parfor
20: distances[0] = parReduceFindMaxDist(distances)
21: if  $n \rightarrow radius < distances[0]$  then
22:   isUpdated  $\leftarrow$  True
23:    $n.radius \leftarrow (n.radius + distances[0])/2$ 
24:    $n.center \leftarrow n.center + ((distances[0] - n.radius)/2)*|\vec{v}|$ 
25:   //  $|\vec{v}|$  is a unit vector from  $n.center$  to the farthest point
26: end if
27: end while

```

---

## VI Conclusion

In this dissertation, we investigate the problem of leveraging emerging hardware to improve the performance of the data analytics frameworks. There are various types of data analytics frameworks with different characteristics and target workloads. To improve the performance of the various frameworks, we enhance the common ground with emerging hardware.

We first design and implement EclipseMR, a novel MapReduce framework with a distributed in-memory cache for data-intensive applications. We design a robust and scalable DHT-based file system. We adopt a distributed in-memory cache that adapts its boundary to improve the load balance and the probability of data reuse. We propose a LAF job scheduling policy for consistent hashing. Our experimental study shows that each components of EclipseMR contributes to enhancing the performance of the MapReduce framework. EclipseMR outperforms Hadoop and Spark for various applications in our evaluation including iterative applications.

To leverage persistent memory in the data analytics frameworks, we propose  $B^3$ -tree, a B+-tree variant that combines the strength of in-memory binary tree and block-based B+-tree while guaranteeing failure-atomicity without explicit logging.  $B^3$ -tree employs the append-only update strategy with binary tree structure inside a node so that the order of the keys can be preserved without expensive sorting and the number of memory fence and cacheline flush instructions can be reduced. As  $B^3$ -tree is based on B+-tree, it can take advantage of balanced tree height and cache-awareness. We can also exploit the sibling pointer of B+-tree to split or merge tree nodes atomically. Our performance study shows that  $B^3$ -tree outperforms the state-of-the-art persistent index - wB+-tree by a large margin, and it shows a comparable performance with selective persistent FPtree.

For the latency-critical applications, we develop Cache-Conscious Extendible Hashing (CCEH) scheme, a variant of extendible hashing, that benefits from the cacheline-sized buckets for byte-addressable persistent memory while providing failure-atomicity. An intermediate layer between the directory and the bucket called segment is introduced to reduce the size of the directory and find a record with minimal cacheline accesses. Our evaluation shows that CCEH successfully eliminates the full-time rehashing overhead so that it can provide two-thirds of query latency compared to the state-of-the-art level hashing scheme.

Lastly, we develop a data parallel tree traversal algorithm for k-nearest neighbor query processing on the GPU, *Parallel Scan and Backtrack* (PSB). PSB algorithm successfully traverses multi-dimensional SS-tree without tiny runtime stack problem and warp divergence. By linearly scanning the relevant leaf nodes, PSB algorithm can benefit from contiguous memory blocks while increasing the chances to optimize SIMD units. Our experiments show that the PSB algorithm outperforms the branch-and-bound kNN query processing algorithm.

## References

- [1] J. Rao and K. A. Ross, “Cache conscious indexing for decision-support in main memory,” in *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.
- [2] ———, “Making B+-trees cache conscious in main memory,” in *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2000.
- [3] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, “FAST: Fast architecture sensitive tree search on modern CPUs and GPUs,” in *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2010.
- [4] J. Ritter, “Graphics gems.” Academic Press Professional, Inc., 1990, ch. An Efficient Bounding Sphere, pp. 301–303.
- [5] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz, “Querying very large multi-dimensional datasets in ADR,” in *Proceedings of the ACM/IEEE SC1999 Conference*, 1999.
- [6] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz, “DataCutter: Middleware for filtering very large scientific datasets on archival storage systems,” in *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, Mar. 2000, pp. 119–133.
- [7] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.
- [8] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2004.
- [9] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein, “MapReduce Online,” in *the 7th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2010.
- [10] L. Popa, M. Budiu, Y. Yu, and M. Isard, “Dryadinc: Reusing work in large-scale computations,” in *Proceedings of the 2009 USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2009.

- [11] S. Sakr, A. Liu, and A. G. Fayoumi, “The family of MapReduce and large-scale data processing systems,” *ACM Computing Surveys*, vol. 46, no. 1, pp. 11:1–11:44, 2013.
- [12] A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat, “M3R: Increased performance for in-memory Hadoop jobs,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 12, pp. 1736–1747, 2012.
- [13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proceedings of the 2010 USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*, 2010.
- [14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.
- [15] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, “Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling,” in *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010.
- [16] Y. Zhao and J. Wu, “Dache: A data aware caching for big-data applications using the MapReduce framework,” in *Proceedings of INFOCOM*, 2013, pp. 271–282.
- [17] M. W. Rahman, X. Lu, N. S. Islam, and D. K. Panda, “HOMR: A hybrid approach to exploit maximum overlapping in MapReduce over high performance interconnects,” in *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*, 2014.
- [18] D. Tiwari and Y. Solihin, “MapReuse: Reusing computation in an in-memory MapReduce system,” in *28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.
- [19] I. Elghandour and A. Abounaga, “ReStore: Reusing results of MapReduce jobs,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 5, no. 6, pp. 586–597, 2012.
- [20] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, “Sharing across multiple MapReduce jobs,” *ACM Transactions on Database Systems*, vol. 39, no. 2, pp. 12:1–12:46, 2014.
- [21] Y. A. Liu, S. D. Stoller, and T. Teitelbaum, “Static caching for incremental computation,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 3, pp. 546–585, 1998.
- [22] W. Pugh and T. Teitelbaum, “Incremental computation via function caching,” in *the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1989.

- [23] J. Arulraj, A. Pavlo, and S. R. Dulloor, “Let’s talk about storage & recovery methods for non-volatile memory database systems,” in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 707–722.
- [24] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Optimistic crash consistency,” in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.
- [25] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, “High performance database logging using storage class memory,” in *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, 2011, pp. 1221–1231.
- [26] W.-H. Kim, B. Nam, D. Park, and Y. Won, “Resolving journaling of journal anomaly in Android I/O: Multi-version B-tree with lazy split,” in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.
- [27] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, “NVWAL: Exploiting NVRAM in write-ahead logging,” in *Proceedings of 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.
- [28] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, “High-performance transactions for persistent memories,” in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 399–411.
- [29] E. Lee, H. Bahn, and S. H. Noh, “Unioning of the buffer cache and journaling layers with non-volatile memory,” in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.
- [30] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won, “WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly,” in *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015.
- [31] S. Mittal and J. S. Vetter, “A survey of software techniques for using non-volatile memories for storage and main memory systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, Jun. 2015.
- [32] S. Mittal, J. S. Vetter, and D. Li, “A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1524–1537, Jun. 2015.
- [33] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, “Failure-atomic slotted paging for persistent memory,” in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

- [34] K. Shen, S. Park, and M. Zhu, “Journaling of journal is (almost) free,” in *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2014.
- [35] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, “Consistent and durable data structures for non-volatile byte-addressable memory,” in *Proceedings of the 9th USENIX conference on File and Storage Technologies (FAST)*, 2011.
- [36] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, “Barrier-Enabled IO Stack for Flash Storage ,” in *Proceedings of the 11th USENIX Conference on File and Storage (FAST)*, 2018.
- [37] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, “NV-Tree: reducing consistency const for NVM-based single level systems,” in *Proceedings of the 13th USENIX conference on File and Storage Technologies (FAST)*, 2015.
- [38] Y. Zhang and S. Swanson, “A study of application performance with non-volatile main memory,” in *Proceedings of the 31st International Conference on Massive Storage Systems (MSST)*, 2015.
- [39] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, “WORT: Write optimal radix tree for persistent memory storage systems,” in *Proceedings of the 15th USENIX conference on File and Storage Technologies (FAST)*, 2017.
- [40] S. Chen and Q. Jin, “Persistent B+-Trees in non-volatile main memory,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 8, no. 7, pp. 786–797, 2015.
- [41] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, “FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory,” in *Proceedings of 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2016.
- [42] G. D. Knott, “Expandable open addressing hash table storage and retrieval,” in *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*. ACM, 1971, pp. 187–206.
- [43] P.-Å. Larson, “Dynamic hashing,” *BIT Numerical Mathematics*, vol. 18, no. 2, pp. 184–201, 1978.
- [44] W. Litwin, “Virtual hashing: A dynamically changing hashing,” in *Proceedings of the fourth international conference on Very Large Data Bases-Volume 4*. VLDB Endowment, 1978, pp. 517–523.
- [45] Oracle, “Architectural Overview of the Oracle ZFS Storage Appliance,” 2018, <https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-001-architecture-overview-zfsa-2099942.pdf>.



- [46] S. Patil and G. A. Gibson, “Scale and concurrency of giga+: File system directories with millions of files.” in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, vol. 11, 2011, pp. 13–13.
- [47] F. B. Schmuck and R. L. Haskin, “Gpfs: A shared-disk file system for large computing clusters.” in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, vol. 2, no. 19, 2002.
- [48] S. R. Soltis, T. M. Ruwart, and M. T. O’Keefe, “The global file system,” in *5th NASA Goddard Conference on Mass Storage Systems and Technologies*, vol. 2, College Park, Maryland, 1996, pp. 319—342.
- [49] S. Whitehouse, “The gfs2 filesystem,” in *Proceedings of the Linux Symposium*. Citeseer, 2007, pp. 253–259.
- [50] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, “Extendible hashing - a fast access method for dynamic files,” *ACM Trans. Database Syst.*, vol. 4, no. 3, Sep. 1979.
- [51] A. Silberschatz, H. Korth, and S. Sudarshan, *Database Systems Concepts*. McGraw-Hill, 2005.
- [52] H. Mendelson, “Analysis of extendible hashing,” *IEEE Transactions on Software Engineering*, no. 6, pp. 611–619, 1982.
- [53] P. Zuo, Y. Hua, and J. Wu, “Write-optimized and high-performance hashing index scheme for persistent memory,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018.
- [54] P. Zuo and Y. Hua, “A write-friendly hashing scheme for non-volatile memory systems,” in *Proceedings of the 33rd International Conference on Massive Storage Systems and Technology (MSST)*, 2017.
- [55] B. Debnath, A. Haghdoost, A. Kaday, M. G. Khatib, and C. Ungureanu, “Revisiting hash table design for phase change memory,” in *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, ser. INFLOW ’15, 2015, pp. 1:1–1:9.
- [56] R. Pagh and F. F. Rodler, “Cuckoo hashing,” *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.
- [57] T. Foley and J. Sugerman, “KD-tree acceleration structures for a gpu raytracer,” in *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005.
- [58] B. Smits, “Efficiency issues for ray tracing,” *Journal of Graphics Tools*, vol. 3, no. 2, pp. 1–14, 1998.

- [59] V. Havran, J. Bittner, and J. Zara, “Ray tracing with rope trees,” in *14th Spring Conference on Computer Graphics (SCCG)*, 1998.
- [60] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan, “Interactive k-d tree gpu raytracing,” in *Symposium on Interactive 3D Graphics and Games (I3D)*, 2007.
- [61] M. Hapala, T. Davidov, I. Wald, V. Havran, and P. Slusallek, “Efficient stack-less bvh traversal for ray tracing,” in *the 27th Spring Conference on Computer Graphics (SCCG ’11)*, 2011.
- [62] J. Kim, W.-K. Jeong, and B. Nam, “Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2258–2271, 2015.
- [63] D. A. White and R. Jain, “Similarity indexing with the SS-tree,” in *Proceedings of the 12th International Conference on Data Engineering (ICDE)*, 1996, pp. 516–523.
- [64] N. Roussopoulos, S. Kelley, and F. Vincent, “Nearest neighbor queries,” in *Proceedings of 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1995, pp. 71–79.
- [65] N. Katayama and S. Satoh, “The SR-tree: An index structure for high-dimensional nearest neighbor queries,” in *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, May 1997, pp. 369–380. [Online]. Available: <http://www.dbl.nii.ac.jp/~katayama/homepage/research/srtree/>
- [66] G. R. Hjaltason and H. Samet, “Distance browsing in spatial databases,” *ACM Transactions on Database Systems*, vol. 24, no. 2, pp. 265–318, Jun. 1999.
- [67] A. Gupta, B. Liskov, and R. Rodrigues, “One hop lookups for peer-to-peer overlays,” in *Ninth Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, May 2003, pp. 7–12.
- [68] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, “Chord: A scalable Peer-To-Peer lookup service for internet applications,” in *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001, pp. 149–160.
- [69] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, “Relational query co-processing on graphics processors,” *ACM Transactions on Database Systems*, vol. 34, no. 4, pp. 21–39, 2009.
- [70] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, “Nectar: Automatic management of data and computation in datacenters,” in *9th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*, 2010, pp. 75–88.

- [71] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, “Tarazu: Optimizing mapreduce on heterogeneous clusters,” in *17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.
- [72] —, “ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters,” in *USENIX Annual Technical Conference*, 2014.
- [73] Y. Guo, J. Rao, and X. Zhou, “iShuffle: Improving hadoop performance with shuffle-on-write,” in *10th USENIX International Conference on Autonomic Computing (ICAC)*, 2013, pp. 107–117.
- [74] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, “Hadoop acceleration through network levitated merge,” in *Proceedings of the ACM/IEEE SC2011 Conference*, 2011.
- [75] “HiBench,” <https://github.com/intel-hadoop/HiBench>.
- [76] R. Appuswamy, C. Gkantsidis, D. Narayana, O. Hodson, and A. Rowstron, “Scale-up vs scale-out for Hadoop: Time to rethink?” in *4th annual Symposium on Cloud Computing (SOCC)*, 2013.
- [77] K. Elmeleegy, “Piranha: Optimizing short jobs in Hadoop,” *Proceedings of the VLDB Endowment (PVLDB)*, vol. 6, no. 11, pp. 985–996, 2013.
- [78] W. Kim, Y. ri Choi, and B. Nam, “Mitigating YARN container overhead with input splits,” in *17th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2017.
- [79] —, “Coalescing HDFS blocks to avoid recurring yarn container overhead,” in *10th International Conference on Cloud Computing (IEEE Cloud)*, 2017.
- [80] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, “Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Trees,” in *Proceedings of the 11th USENIX Conference on File and Storage (FAST)*, 2018.
- [81] P. L. Lehman and S. B. Yao, “Efficient locking for concurrent operations on B-trees,” *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 650–670, 1981.
- [82] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li, “Quartz: A lightweight performance emulator for persistent memory software,” in *Proceedings of the 15th Annual Middleware Conference (Middleware ’15)*, 2015.
- [83] H. E. Lab, “Quartz,” 2018, <https://github.com/HewlettPackard/quartz>.
- [84] J. Huang, K. Schwan, and M. K. Qureshi, “Nvram-aware logging in transaction systems,” *Proceedings of the VLDB Endowment*, vol. 8, no. 4, 2014.

- [85] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [86] C. S. Ellis, “Extendible hashing for concurrent operations and distributed data,” in *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ser. PODS ’83. New York, NY, USA: ACM, 1983, pp. 106–116. [Online]. Available: <http://doi.acm.org/10.1145/588058.588072>
- [87] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2002.
- [88] O. Shalev and N. Shavit, “Split-ordered lists: Lock-free extensible hash tables,” *J. ACM*, vol. 53, no. 3, pp. 379–405, May 2006.
- [89] B. Goetz, “Building a better HashMap: How ConcurrentHashMap offers higher concurrency without compromising thread safety,” 2003, <https://www.ibm.com/developerworks/java/library/j-jtp08223/>.
- [90] Intel, “Intel Threading Building Blocks Developer Reference ,” 2018, <https://software.intel.com/en-us/tbb-reference-manual>.
- [91] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman, “Algorithmic improvements for fast concurrent cuckoo hashing,” in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 27.
- [92] Oracle, “Java Platform, Standard Edition 7 API Specification,” 2018, <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html>.
- [93] J. L. Carter and M. N. Wegman, “Universal classes of hash functions (extended abstract),” in *Proceedings of the ACM 9th Symposium on Theory of Computing (STOC)*, 1977, pp. 106–112.
- [94] K. A. Ross, “Efficient hash probes on modern processors,” in *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*, 2007.
- [95] I. Kamel and C. Faloutsos, “On Packing R-trees,” in *Proceedings of the second international conference on Information and knowledge management*, 1993, pp. 490–499.
- [96] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, “Analysis of the clustering properties of the hilbert space-filling curve,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, no. 1, pp. 124–141, 2001.

- [97] NVIDIA, “Thrust,” <https://developer.nvidia.com/thrust>.
- [98] K. V. Mardia, J. Kent, and J. M. Bibby, *Multivariate Analysis*. Academic Press, 1979.
- [99] N. Megiddo, “Linear-time algorithms for linear programming in  $r^3$  and related problems,” in *23rd Annual Symposium on Foundations of Computer Science*, 1982, pp. 329–338.
- [100] M. Gordon, “Parallel computation of minimal enclosing circle using MPI and OpenMP,” <http://www-personal.umich.edu/msgsss/mec/mec.pdf>.
- [101] M. Karlsson, O. Winberg, and T. Larsson, “Parallel construction of bounding volumes,” in *Proceedings of the Annual Swedish Computer Graphics Association Conference.*, 2010.
- [102] K. Fischer, B. Gärtner, and M. Kutz, “Fast smallest-enclosing-ball computation in high dimensions,” in *Algorithms-ESA 2003*. Springer, 2003, pp. 630–641.
- [103] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft, “When is nearest neighbor meaningful?” in *Proceedings of 7th International Conference on Database Theory (ICDT)*, 1999.
- [104] S. Brown, “Superfast Nearest Neighbor Searches using a Minimal KD-tree,” 2010, gPU Technology Conference (GTC 2010) (<http://www.gpupotechconf.com>).

## Acknowledgements

I would like to first express my deep appreciation to Prof. Beomseok Nam and Prof. Young-ri Choi, my advisors, for their guidance, encouragement, and advice. I couldn't make it this far without their support. I would also like to thank my thesis committees, Prof. Sam H. Noh, Prof. Woongki Baek, and Prof. Won-ki Jeong, for their valuable critiques of this work.

I am lucky to have good friends and colleagues who are always willing to help me and share their time. I would like to thank Wookhee Kim for devoting him to the laboratory and always helping me. I would also like to thank Eunji Hwang for always kindly treating me. I appreciate my other good friends in CISSR group for their time and support.

My grateful thanks are extended to my dear friends who have been together for a long time – Woohyuk Choi, Imdo Jeong, Inwoo Hwang, Soyoung Park, Seil Jeong, Yesung Kang, Dasom Jeong, Hyungsuk Choi, Muyoung Lee, Sumin Hong, Byeongju Han, Seontae Kim, Kyu-Yul Lee, and many more.

I am also very fortunate to have my lover, Jin Yeong Kim. She has provided the emotional support that I needed to overcome all the difficulties I had. She has always been there for me so that I can get through it. Without her, I could not have done all of this.

Finally, I am very grateful for all the love and support of my family, Girak Nam, Mija Lee, and Joonghyeon Nam.

