BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B$^+$-trees

by

Amanda Li

B.S. Computer Science and Engineering, Mathematics
Massachusetts Institute of Technology, 2022

Submitted
to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

Author .....
Department of Electrical Engineering and Computer Science
January 20, 2023

Certified by .....
Helen Xu
Postdoctoral Scholar
Lawrence Berkeley National Laboratory
Thesis Supervisor

Certified by .....
Charles E. Leiserson
Professor of Computer Science and Engineering
Massachusetts Institute of Technology
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

BP-Tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B$^+$-trees

by

Amanda Li

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents the ***BP-tree***, an efficient concurrent key-value store based on the B$^+$-tree, that uses large leaf nodes to optimize for range-query performance without sacrificing update speed by using large leaf nodes. B$^+$-trees are a fundamental data structure for implementing in-memory indexes in databases and storage systems. B$^+$-trees support both point operations (i.e., inserts and finds) and range operations (i.e., iterators and maps). There is an inherent tradeoff between point and range operations however, since the optimal node size for point operations is much smaller than the optimal node size for range operations. To avoid any slowdown in point operations, this thesis introduces a novel insert-optimized array called the ***buffered partitioned array*** (BPA) to efficiently organize data in leaf nodes.

Using the buffered partitioned array, the BP-tree overcomes the decades-old tradeoff between point and range operations in B$^+$-trees. Experiments show that on 48 hyper-threads, the BP-tree supports slightly faster (by about 1.1×) point operations than the best-case configuration for B$^+$-trees for point operations while supporting between 1.4×–1.7× faster range operations. On workloads generated from the Yahoo! Cloud Serving Benchmark (YCSB), the BP-tree is faster (by about 1.1×) on all point operation workloads compared to the B$^+$-tree, and slower (by about 1.15×) on the short range operation workload compared to the B$^+$-tree. Furthermore, this work extends the YCSB to include large scan and map workloads, commonly found in database applications, and find that the BP-tree is between 1.2×–1.4× faster than the B$^+$-tree on these workloads. This thesis contains my joint work with Helen Xu, Brian Wheatman, Manoj Marneni, and Prashant Pandey.

Thesis Supervisor: Helen Xu
Title: Postdoctoral Scholar

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

Acknowledgments

I would like to thank Helen Xu for mentoring me over the past two years. Throughout the development of this research, you have been incredibly supportive and have provided invaluable technical and personal advice. The passion you have for your work is infectious and I am very grateful for your constant encouragement.

I would like to thank Charles E. Leiserson for inspiring me to pursue this field of research and his guidance throughout my thesis. Your teachings, as both my professor and research advisor, have deeply influenced my perspective on computer science and my path at MIT.

I would also like to thank Brian Wheatman, Manoj Marneni, and Prashant Pandey for their insights and support in this joint effort. Thank you for sharing your knowledge and providing extremely helpful feedback during our weekly meetings.

Lastly, I would like to thank my friends and family for all their love and support.

This research was sponsored in part by the United States Air Force Research Laboratory under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein. This research was also supported in part by Los Alamos National Laboratory (LANL) under Subcontract Number 531711.

Contents

Chapter 1

Introduction

This thesis shows how to build an efficient key-value store that outperforms the traditional B$^+$-tree on range operations, while maintaining performance for point operations.

The B-tree [5] has been the fundamental access path structure in databases and storage systems for over five decades [14, 23], and the B$^+$-tree is a widely implemented variant of the B$^+$-tree in real-world applications. B-trees are an extension of self-balancing binary search trees to arbitrary fanouts (with more than two children per node). They store elements in each node in a sorted array. A B$^+$-tree is a scan-optimized variant of B-trees that stores all data records in the leaves and only pivot keys in the internal nodes. Given a cache-line size Z, a B$^+$-tree with N elements and node size $B = \Theta(Z)$ supports the point operations `insert` and `find` in $O(\log_B(N))$ cache-line transfers in the I/O model [1]. B$^+$-trees are one of the top choices for in-memory indexing [65] due to their cache efficiency though they were initially introduced for indexing data stored on disk [5]. This thesis presents my joint work with Helen Xu, Brian Wheatman, Manoj Marneni, and Prashant Pandey in improving the performance of in-memory B$^+$-trees.

B$^+$-trees are especially popular in databases and file systems because they support logarithmic point operations (inserts and finds) and efficient range operations (range queries and scans) that read blocks of data [32, 57]. They are also extensively used as the in-memory index in many popular databases such as MongoDB [40], CouchDB [17], ScyllaDB [59], PostgreSQL [51], and SplinterDB [15].

B$^+$-trees support updates as well as member, predecessor, and successor queries in

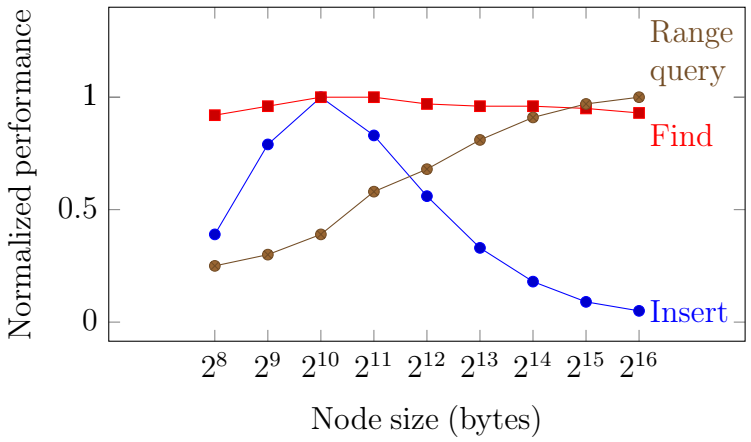

Figure 1-1: Normalized performance compared to the fastest configuration (for each operation) in a concurrent B$^+$-tree with 48 hyperthreads. The x-axis represents the node size of the B$^+$-tree in bytes. The y-axis represents the throughput for a given configuration relative to the throughput for the best-case configuration for a given operation (1.0 is the best possible value). Note that while only node sizes of powers of 2 are plotted, we ran additional experiments on other node sizes and found no anomalies between powers and non-powers of 2 node sizes.

a logarithmic number of cache-line transfers. In a comparison based I/O model, this number of cache-line transfers is the best possible for member queries [10]. This thesis focuses on the case where $B = \Theta(Z)$.

B$^+$-trees also support range operations that scan across the leaf level of the tree. For range queries, the B$^+$-tree requires $O(\log_B(N) + k/B)$ cache-line transfers, where k is the number of elements in the range. For long range queries, the k/B factor is the higher-order term, so increasing the node size improves range-query performance. The performance of point operations does not improve however and may even suffer with larger nodes. Therefore, there is an inherent tradeoff between point operations (insert and query) and range queries in the B$^+$-tree as the optimal node size is quite different for point and range operations.

As illustrated in Figure 1-1, the B$^+$-tree exhibits[1] a tradeoff between point and range operations depending on the selection of node size. Prior work showed that setting the node size much larger than the cache-line size can improve both point and range operations [13, 26]. Similarly, we found that the optimal node size for point operations was 2^{10} bytes in the tested B$^+$-tree. In contrast, the range-query performance improves with the node size as the nodes grow past 2^{10} bytes in size, but it starts to stagnate

[1] Chapter 6 contains all details about the experimental setup and method.

beyond 2^{16}. Large nodes improve range queries because they reduce cache misses by reading more contiguous data. As the node size grows however, insertion performance suffers because more elements are shifted around upon each insert.

Point and range operations in key-value stores. One of the core use cases for B^+-trees is to support key-value (KV) stores [39], a ubiquitous method of storing data as a collection of records, or key-value pairs. KV stores are used extensively in systems such as Dynamo [18], Redis [31], and Memcached [19, 48].

KV stores have traditionally been optimized and benchmarked for point operations (e.g., get and put) that underlie online transaction processing (OLTP) applications such as those in the Yahoo! Cloud Serving Benchmark (YCSB) [16]. The original YCSB workloads center around point operations such as point insertions and queries. They contain range queries, but only in a limited capacity because range operations are not as common as point operations in OLTP.

Emerging applications increasingly require both fast point and range operations (e.g., range queries and maps) in the same KV store to enable integrated support for both transactional and analytic processing [50, 55, 25]. Range queries from transactional workloads are often short (i.e., they involve only a few elements)—the default configuration in YCSB generates range queries with a maximum length of 100. In contrast, range queries from analytical workloads may be much longer and access a constant fraction of the data (e.g., 1% or 10%) [50]. Real-time analytics require fast range operations to process both real-time data and archived data as quickly as possible [9, 12, 20]. Bioinformatics applications also use range queries to investigate genomic data [24, 62, 45]. Graph analytics workloads often require quickly scanning through all the neighbors of a given source node [46].

Systems optimized for either point operations or range operations may suffer on the other type of workload. For example, prior work showed that state-of-the-art KV stores such as Cassandra [11], RAMCloud [43], and RocksDB [56] perform poorly on long range queries because they were designed for point and short queries [50, 49]. Furthermore, HBase [27] integrates support for point and range operations but has been shown to underperform on point lookups compared to other KV stores.

Overcoming the point-range tradeoff in B$^+$-trees. The goal of this thesis is to overcome the inherent point-range tradeoff in B$^+$-trees by making the nodes bigger to support fast range operations without compromising on point-insert performance. As shown in Figure 1-1, simply making the nodes bigger while keeping the sorted array data structure in the nodes does not solve the problem because it improves range operations at the cost of point operation throughput.

This work argues that not all nodes in a B$^+$-tree need to be of the same size. Leaf nodes contain all of the data records and making them large results in higher locality and faster range scans. Internal nodes only store pivots and keeping them small results in faster updates.

Making the leaf nodes larger beyond a certain point, however, requires organizing data records inside leaf nodes to support efficient updates. As we shall see in Chapter 3, naively increasing the size of only the leaves while keeping the sorted arrays in the leaves does not solve the problem because it improves range operation throughput at the cost of point operation throughput.

To improve range operation performance while maintaining fast point operations, we introduce a new insert-optimized array-based data structure called the ***buffered partitioned array*** (BPA) that we incorporate into B$^+$-tree nodes to allow them to grow in size without sacrificing point-insert throughput. The BPA is faster for inserts than a traditional array for two main reasons. First, it buffers insertions to avoid shifting elements on every insert, drawing inspiration from write-optimized data structures [22]. Second, it partitions the array into blocks and leaves empty spaces in the blocks to further avoid shifting during every insert, drawing inspiration from the Packed Memory Array (PMA) data structure [30, 7].

We use the BPA to create the ***BP-tree***, a variant of the traditional B$^+$-tree that uses the BPA in the leaves and sets the leaf size to be much larger than the internal node size. The BP-tree is optimized for long range queries that traverse multiple leaves in the B$^+$-tree. It improves long range queries by avoiding pointer indirections that would have occurred with smaller leaf nodes thereby converting random reads into sequential ones. Furthermore, the insert-optimized BPA ensures that there is no impact on the perfor-

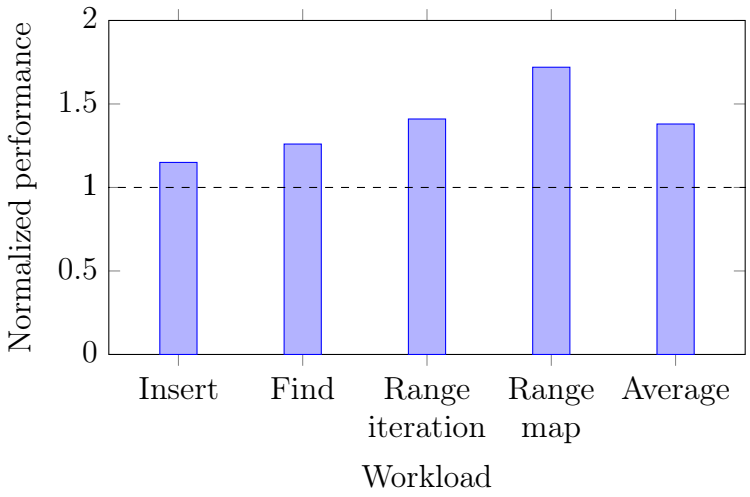

Figure 1-2: Relative speedup of the BP-tree compared to the baseline B+-tree on a series of microbenchmarks. The x-axis lists the evaluated workloads. The y-axis represents the throughput of the BP-tree relative to the throughput of the B$^+$-tree (i.e. above 1 means the BP-tree is faster than the B$^+$-tree).

mance of point operations. In fact, it speeds up the point operations in some benchmarks.

Results summary. We implemented a concurrent BP-tree using the state-of-the-art TLX B$^+$-tree [8] and an optimistic concurrency control scheme [33]. We use the concurrent version of the TLX B$^+$-tree with 1024-byte nodes as the baseline because this is the best case for point inserts.

Figure 1-2 demonstrates that the BP-tree improves the performance of range iteration by up to 1.4× and range maps by up to 1.7× without giving up point operation performance when compared to the best-case configuration for point operations in a concurrent B$^+$-tree (B=1024, from Figure 1-1). Since different use cases may require different types of range operations (iteration or maps), we include both in the evaluation. The BPA enables the BP-tree to support faster range operations with much larger[2] leaves. Furthermore, the BP-tree is slightly faster than the baseline for both point inserts and queries (finds) even though the leaves in the BP-tree are almost 8× larger than those in the baseline.

In addition to the baseline B$^+$-tree, Figure 1-3 compares the range and point operation performance of different node size configurations of the plain B$^+$-tree to that of the BP-tree. This shows that while some configurations of the plain B$^+$-tree can exceed the performance of the BP-tree in a single dimension, the BP-tree achieves higher

[2]The default configuration for the BP-tree sets $B_{internal}$=1024 bytes and B_{leaf}=8704 bytes.

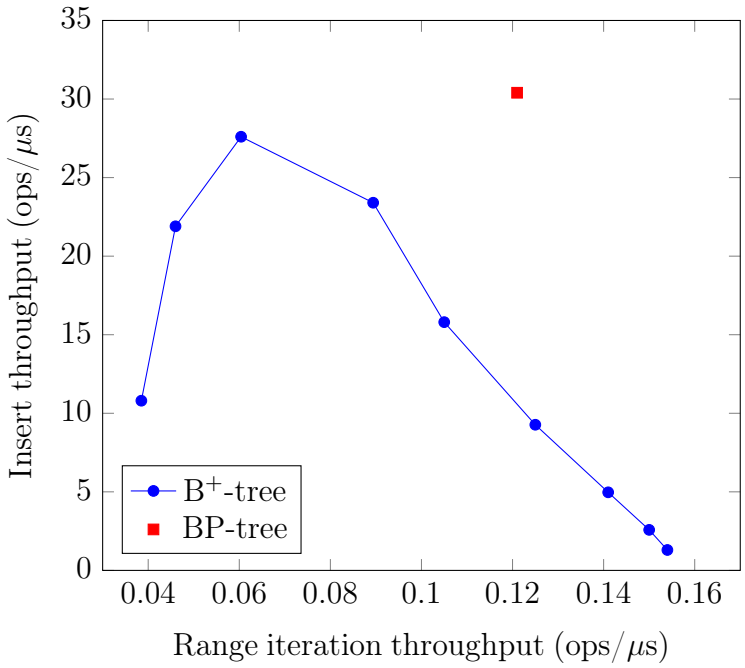

Figure 1-3: Throughput of range iteration (maximum length of 100k) versus throughput of inserts at varying node sizes for the B$^+$-tree, compared to the BP-tree. Each point plotted represents the range-iteration and insert performance of one configuration of the B$^+$-tree or the BP-tree.

performance when considering both range and point operations.

We also tested the baseline concurrent B$^+$-tree and BP-tree on default workloads from YCSB and report the results in Figure 1-4. The BP-tree achieves slightly faster performance (about 1.1×) on all point operation workloads compared to the B$^+$-tree, and slower performance (within 1.15×) on the short range operation workload compared to the B$^+$-tree. Since the BP-tree is optimized for long range queries, we created two new workloads using the YCSB generator. The first one is called workload X and contains large range iterations. The second one is called workload Y and contains large range maps. The BP-tree is about 1.2× faster on workload X and 1.4× faster on workload Y when compared to the baseline.

Types of range operations. Traditional B$^+$-trees (and most key-value stores) store elements in globally sorted order and implement range queries with ordered iteration over elements by key.

The importance of ordered iteration within a range query depends on the use case. For example, YCSB requires that the results of a range query can be iterated through

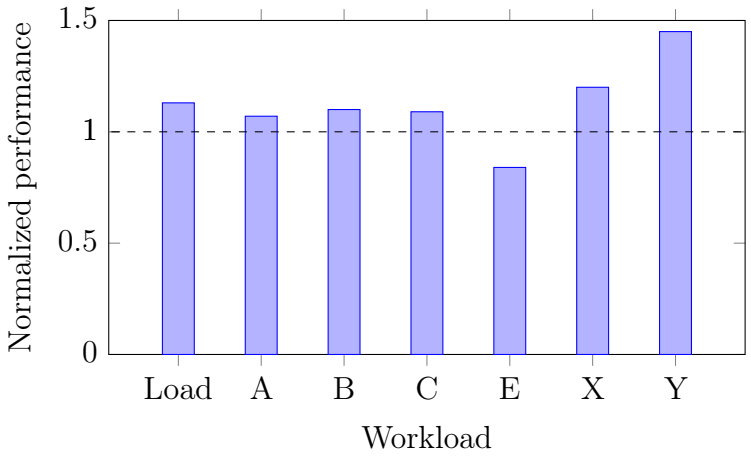

Figure 1-4: Relative speedup of the BP-tree compared to the baseline B+-tree on workloads generated from YCSB. The x-axis lists the evaluated workloads. The y-axis represents the throughput of the BP-tree relative to the throughput of the B^+-tree (i.e. above 1 means the BP-tree is faster than the B^+-tree).

in sorted order to simulate an application example of threaded conversations. Cassandra [11] supports this type of query natively by taking as input a start key and a length (i.e., the number of consecutive following entries to scan). We refer to this operation as ***range iteration***.

On the other hand, some applications may not necessarily need to access the keys in order. Some examples include graph-processing systems [4, 60], feature storage for machine learning [28, 37, 47], and file system metadata management [54, 57]. In contrast to Cassandra's range query API, HBase's range query API [27] takes as input an interval of start and end keys and scans entries with keys in the interval. We refer to this operation as ***range map***.

We implement and evaluate both range iteration, which iterates over the requested number of keys in order, and range map, which maps over an interval of keys but not necessarily in order.

Contributions

Specifically, the thesis' contributions are as follows:

- An empirical evaluation of the impact of the node size on various B^+-tree operations in a concurrent setting.

- The design and implementation (in `C++`) of the buffered partitioned array (BPA), an insert-optimized data structure that reduces element moves compared to a sorted array.
- The design and implementation (in `C++`) of the BP-tree, a variant of the B$^+$-tree incorporating the BPA in the leaves, which overcomes the decades-old point-range operation tradeoff in B$^+$-trees.
- An evaluation of BP-tree compared to a traditional B$^+$-tree on microbenchmarks and workloads from YCSB that demonstrates that BP-tree supports faster range operations without sacrificing point operation performance.

Map. The rest of the thesis is organized as follows. Chapter 2 presents necessary preliminaries about B$^+$-trees and concurrency schemes required to understand the data structures and implementations in this thesis. Chapter 3 demonstrates the need for leaf-specific data structure design to overcome the point-range tradeoff because simply increasing the leaf size does not solve the issue. Chapter 4 describes the buffered partitioned array (BPA) data structure that we use to enable larger B$^+$-tree leaf nodes while maintaining fast point operations. Chapter 5 shows how to incorporate the BPA into a B$^+$-tree to create the BP-tree. Chapter 6 empirically compares the BP-tree with a baseline B+-tree on a suite of microbenchmarks and on workloads from YCSB. Chapter 7 surveys related work, and Chapter 8 provides concluding remarks.

Chapter 2

Background

This chapter provides necessary background to understand the data structure improvements in this thesis. First, we introduce the classical "Disk-Access Machine (DAM) model" and a refinement to the DAM model called the "affine model" [6]. We use the DAM model to measure the cost of B$^+$-tree operations, and in Chapter 5, use the affine model to explain the empirical differences between data structures. Next, we review details about B$^+$-trees and their operations, which we will build on in the BP-tree. Finally, we describe the concurrency control mechanism in the concurrent B$^+$-tree that we use as a baseline and as a starting point for the BP-tree implementation.

Memory models

The ***Disk-Access Machine (DAM) model*** [1] captures algorithm cost in hierarchical memory by taking into account non-uniform access times in different levels of memory. It models two levels of memory: a small fixed-size cache and an unbounded-size slow memory. Any data must first be in the cache before it processed. Data is transferred between the two levels in cache lines of size Z. An algorithm's cost is measured in cache-line transfers.

The ***affine model*** [6, 3, 58] refines the DAM model to explicitly account for the cost of random vs contiguous memory access. In the affine model, an I/O of x words has cost $1+\alpha x$, where $\alpha \ll 1$ is a hardware parameter. The 1 represents the normalized setup

cost of doing an indirection (or seek, on disk) and α is the normalized bandwidth cost.

B$^+$-tree design and operations

B$^+$-trees generalize self-balancing binary trees to arbitrary fanouts to take advantage of the speed of contiguous memory access [5]. Just like binary trees, B$^+$-trees store elements in sorted order. Traditionally, B$^+$-trees store $B = \Theta(Z)$ elements per node in a sorted contiguous array. The height of a B$^+$-tree with N elements and node size B is $O(\log_B(N))$.

A B$^+$-tree is a scan-optimized variant of B-trees that stores all elements in the leaves and replicates some of the elements in the internal nodes. In contrast, traditional B-trees store each element exactly once (either in the internal nodes or leaves). B$^+$-trees support faster scans than B-trees because they can scan over all of the elements without having to access internal nodes.

A B$^+$-tree exposes the following four operations.

`insert(k, v)`

Inserts a key-value pair `(k, v)`.

`find(k)`

Returns a pointer to the element with the smallest key that is at least `k`.

`iterate_range(start, length, f)`

Applies the function `f` to `length` elements in key order, starting with the element with the smallest key that is at least `start`.

`map_range(start, end, f)`

Applies the function `f` to all elements with keys in the range [`start`, `end`).

Point operations refer to `insert` and `find`. Range operations refer to `iterate_range` and `map_range`. We omit the discussion of deletes for simplicity, but they are symmetric to insertions.

We now review the asymptotic bounds for the four main operations on B$^+$-tree with N elements and node size $B = \Theta(Z)$.

The B$^+$-tree supports the point operations `insert` and `find` in $O(\log_B(N))$ cache-line transfers. To find an element in a B$^+$-tree, we traverse the internal nodes and follow the pivots (elements at internal nodes) to find the leaf that the target element might reside in. This procedure takes $O(1)$ cache-line transfers at each level of the tree for a total cost of $O(\log_B(N))$ cache-line transfers. Inserts begin with a `find` for the correct leaf to insert the element into. To maintain elements in sorted order, the B$^+$-tree shifts elements in the array in the target leaf to make space and place the element in the correct position. If the leaf becomes full, it ***splits*** into two leaves and the midpoint is promoted to become a pivot in the internal nodes. This promotion procedure proceeds up the tree recursively, if necessary.

The B$^+$-tree supports the range operations `iterate_range` and `map_range` in $O(\log_B(N)+k/B)$ cache-line transfers where k is the number of elements in the range. A range operation in a B$^+$-tree is comprised of a `find` for the smallest element with a key that is at least `start`, which takes $O(\log_B(N))$ cache-line transfers. Since the B$^+$-tree stores elements in sorted order, it implements both `iterate_range` and `map_range` with a forward scan from the starting element until the end of the range. Since there are $\Theta(Z)$ elements in each node, this scan of k elements takes $\Theta(k/B) = \Theta(k/Z)$ cache-line transfers.

Concurrency control

This section describes the optimistic concurrency control mechanism [33] used in the B$^+$-tree and BP-tree in this thesis to support simultaneous operations (i.e., concurrent inserts, finds, and range queries). Each internal node is locked with a read/write lock, where multiple readers are allowed to access the node concurrently if no write lock is held. Each leaf node is locked with a simple exclusive lock.

The operations below use hand-over-hand locking [29] to traverse the tree. Hand-over-hand locking is a method of grabbing successive locks which first grabs a successor lock (e.g., the lock on the child node) prior to releasing a lock on its predecessor (e.g., the lock on the parent node). We first describe concurrency mechanisms for all B$^+$-tree operations, then argue deadlock freedom and termination.

Insert. Insertions first make an optimistic descent by taking hand-over-hand read locks from the root down to the leaf, and then locking the leaf. If we are able to insert into the leaf without causing a split, then we successfully insert into the leaf and release all locks. If inserting into the leaf would cause a split, we check if the parent can handle an additional element. If the parent can handle the additional insert, the change will not propagate farther up the tree and we just need to acquire the write lock on the parent of the leaf. We first try to upgrade the current read lock on the parent to a write lock, which can only be done if no other threads hold the read lock on the parent. If the lock can be upgraded, we can complete the insert. If the parent of the leaf cannot be upgraded, since some other thread is trying to read or write the parent, we release the lock entirely and restart the insertion operation with a pessimistic second descent. In the second descent, we take write locks from the root down to the leaf. Then we lock the leaf and the new right leaf (from the split), insert into the appropriate leaf, and propagate the midpoints back of the tree, unlocking as we go.

Find. Finds take read locks in a hand-over-hand fashion from the root down to the leaf, then lock the leaf and search for the key within the leaf. The lock on the leaf nodes are exclusive locks.

Range query. Range queries first perform a find on the start key to locate the starting leaf of the query, then take locks from left-to-right as needed in a hand-over-hand fashion starting from the leaf that resulted from the find.

Theorem 1. *The B^+-tree is deadlock free.*

Proof. To prove that the B^+-tree does not deadlock, let us show that the B^+-tree imposes a total resource ordering [41] at every given time-step. Let V be the set of nodes in a B^+-tree. Let us order these resources, i.e. nodes V_i for $i = 1,2,...,|V|$, from top-down, then left-to-right via the bijection $f : V \to [1,|V|]$ at some arbitrary time-step. We refer to $f(V_i)$ as the value of V_i. Let $f(V_i) < f(V_j)$ if V_j is on a lower level than V_i, or if they are on the same level and V_j is to the right of V_i in order of pivots. For example, suppose V_1 is the root node, V_2 is the left-most node of the second level, V_3 is the next node in the second level. Then $f(V_1) < f(V_2) < f(V_3)$, and furthermore $f(V_1) = 1, f(V_2) = 2$, and $f(V_3) = 3$.

Suppose there exists P parallel processes at some arbitrary time. Each process p_i for $i=0,1,...,P-1$ holds a set of resources $S_i \subseteq V$. Let us assign each process the maximum value over the resources it holds. Formally, let p_i have value $w_i \in [0,|V|]$ such that $w_i = 0$ if $|S_i| = 0$, and $w_i = \max(f(s)|s \in S_i)$ otherwise.

Let us now show that each process p_i only waits for resources with higher value than w_i in all operations. Finds and range queries trivially meet this condition, as all resources are acquired from top-down while searching, then left-to-right while querying.

For inserts, there are two cases: inserts that only require the optimistic descent, and inserts that require the second descent. In the optimistic descent and if the insert does not cause a split, resources are trivially acquired top-down. In the second case, some more inspection is needed when inserts would cause a split. As described prior, upon a split at the leaf, the process attempts to upgrade the lock of the leaf's parent. While the parent does have a lower value than the leaf, the process does not wait and instead immediately fails if other threads hold a read lock on the parent. In this case, the process releases all resources and restarts from the root, again acquiring resources top-down. Lastly, splits create a new right node at the same level as the original split node, so processes maintain the left-to-right order of resource acquisition. Thus the B^+-tree imposes a total resource ordering across all operations and does not deadlock.

□

Theorem 2. *The B^+-tree is starvation-free.*

Proof. To prove that the B^+-tree is starvation-free, let us show that each process will only wait for a bounded set of internal nodes that does not grow and a finite number of leaf nodes. Weak fairness of the scheduler is assumed, meaning that when an action is continuously enabled, the action will eventually be executed [35]. We continue to use the notation for resources and processes established in Theorem 1.

A process p_i for all $i=0,1,...,P$ must wait in two cases at a given time-step: either the process must wait to acquire an internal node's write lock, or the process must wait to acquire a leaf's exclusive lock. In the first case, let M_i be the node corresponding to w_i, i.e. the highest value node held by process p_i. Let h_i be the height of M_i (i.e., its

distance from the leaves). For example, the leaves have height 0 and any parent of a leaf has height 1. Then the process p_i waits for at most $h_i - 1$ internal-node descendants of M_i. As described prior, all operations acquire internal nodes in top-down order from the root. In order for other processes to acquire internal-node descendants of M_i, they must first acquire M_i. Thus process p_i waits for a bounded number of internal nodes and will eventually reach the leaf level.

In the second case, the process p_i is waiting for exactly one leaf node at a given time-step. The process is either waiting at the parent of the leaf or the left predecessor (in range queries). All processes at the leaf node release the lock upon either completion or failure (in the optimistic descent of inserts). All processes request finitely many leaf nodes in total. Thus given weak fairness of the scheduler, this process will terminate. □

Chapter 3

The Point-Range Tradeoff

This chapter empirically demonstrates the tradeoff in point-range operations by varying the B$^+$-tree leaf node size. We find that simply increasing the size of the leaves greatly sacrifices point insertion performance for range-operation performance. These experiments establish a need for leaf-specific data structure design in B$^+$-trees beyond sorted arrays in order to enable larger leaf nodes.

Figure 3-1 and Table 3.2 report the performance of a version of the B$^+$-tree that fixes the internal node size at 1024 bytes (to match that of the best-case B$^+$-tree for inserts) and increases the size of the leaf nodes (starting at 1024 bytes). Although increasing the leaf node size can improve range query[1] throughput by almost 4×, the B$^+$-tree with the largest leaf node size we tested is almost 20× slower for inserts than the best-case B$^+$-tree configuration for point insertions. The middle ground of a B$^+$-tree with internal nodes of 1024 bytes and leaf nodes of 4096 bytes that improves range query throughput by about 1.5× is about 2× slower for inserts when compared to the B$^+$-tree with all nodes of size 1024 bytes.

Figures 1-1 and 3-1 exhibit similar trends even though the first one varies the size of all the B$^+$-tree nodes, whereas the second varies only the B$^+$-tree leaf size. Since all the data records in a B$^+$-tree are present in the leaves, the internal node size does not significantly affect overall performance. Internal nodes do not need a specialized data structure for inserts because the internal nodes are updated rarely (especially as the

[1]The plot illustrates range iteration performance, but range map performance is similar in the B$^+$-tree.

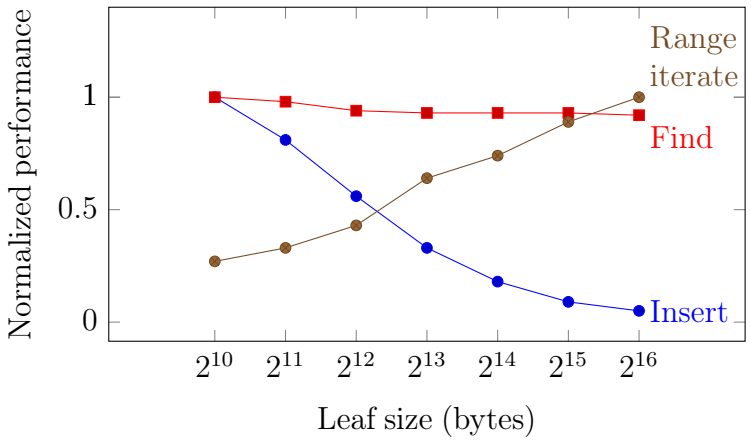

Figure 3-1: Normalized performance compared to the fastest configuration (for each operation) in a concurrent B$^+$-tree with variable leaf size on 48 hyperthreads. The x-axis represents the leaf node size of the B$^+$-tree in bytes. The y-axis represents the throughput for a given configuration relative to the throughput for the best-case configuration for a given operation (1.0 is the best possible value).

B+-tree		*Insert*		*Find*		*Iterate range*		*Map range*	
Internal size (bytes)	*Leaf size (bytes)*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*
1024	1024	2.75E7	1.00	4.22E7	1.00	1.93E9	0.27	1.84E9	0.28
1024	2048	2.23E7	0.81	4.12E7	0.98	2.30E9	0.33	2.12E9	0.32
1024	4096	1.54E7	0.56	3.98E7	0.94	3.02E9	0.43	2.71E9	0.41
1024	8192	9.08E6	0.33	3.92E7	0.93	4.47E9	0.64	4.24E9	0.65
1024	16384	4.93E6	0.18	3.94E7	0.93	5.23E9	0.74	5.14E9	0.78
1024	32768	2.56E6	0.09	3.92E7	0.93	6.24E9	0.89	5.84E9	0.89
1024	65536	1.31E6	0.05	3.87E7	0.92	7.03E9	1.00	6.57E9	1.00

Table 3.2: Throughput and normalized performance of point and range operations of a B$^+$-tree with fixed-size internal nodes but varying leaf nodes. Point operation throughput is reported in operations/s and range query throughput is reported in (expected elements processed)/s. We use N.P. to denote the normalized performance in the B$^+$-tree compared to the best B$^+$-tree configuration for that operation (1.0 is the best possible value).

leaf size grows). Furthermore, the leaf node size determines range-operation throughput because operations in a B$^+$-tree perform a scan in the leaves. B$^+$-trees with larger leaves exhibit better locality during scans and therefore achieve higher range-operation throughput. See the end of Chapter 5 for the theoretical analysis in the affine model of range scans with sequential and random I/O.

Based on the results in this chapter, we focus on designing specialized data structures for B$^+$-tree leaves because their size and data structure choice determine the overall performance of both point and range operations. Especially, the range iteration and range map performance which is critical in numerous database and storage systems workloads.

Chapter 4

The Buffered Partition Array

This chapter describes the buffered partitioned array (BPA) data structure, which enables the BP-tree to maintain large leaf nodes without sacrificing updatability. It then describes how the buffered partitioned array supports the four operations `insert`, `find`, `iterate_range`, and `map_range`, which were presented in Chapter 2:.

The main idea behind the BPA is to create a data structure for B^+-tree nodes that uses a larger contiguous block of memory compared to traditional B^+-tree node sizes to enable fast scans while maintaining fast inserts. As demonstrated in Chapter 3, simply increasing the leaf node size significantly degrades B^+-tree insert performance. Furthermore, the leaf node size determines overall performance because most of the writes affect only the leaves. Therefore, we design a new data structure specifically for B^+-tree leaves.

The BPA improves insertion throughput when compared to a sorted array by reducing data movement in two ways. First, the BPA buffers inserts to amortize data movement across operations. Next, it maintains empty spaces in the data structure in a "blocked structure" to avoid element shifting as much as possible even when the buffer becomes full. Finally, it does not maintain a global sorted order across the array. Specifically, the items inside blocks are not stored in order and that allows new inserts to be placed at the ends of blocks instead of requiring element shifts.

Structure

The BPA uses a contiguous array to store its data, but partitions that array into three sections called the "log" the "header" and the "blocks" as illustrated in Figure 4-1. It is parametrized by the following values:

- `log_size`: the maximum number of buffered inserts.
- `num_blocks`: the number of blocks in the data structure.
- `block_size`: the maximum number of elements per block.

The ***log*** encompasses the first `log_size` slots (i.e., locations [0,`log_size`)) and is used to buffer inserts that will later propagate to the rest of the data structure. The ***header*** uses the next `num_blocks` slots (i.e., locations [`log_size`, `log_size` + `num_blocks`)) to partition the rest of the data structure by range. Each slot in the header holds the minimum element (or a block marker) in the corresponding block. The elements in the header are sorted. Finally, there are `num_blocks` ***blocks*** of `block_size` slots each. Each block's elements fall in the range denoted by the corresponding header element. That is, the elements in block i fall in the range denoted by the elements at positions [`log_size`$+i$,`log_size`$+i+1$). The i-th block's elements start at position `block_start` = `log_size` + `num_blocks` + $i \times$ `block_size`. The i-th block encompasses the cells in positions [`block_start`,`block_start` + `block_size`). In contrast to the elements in the header, elements in the log and each individual block may not be sorted.

Just as in other buffered data structures such as the B^ε-tree [22], there may be two copies of an element in a BPA: one in the log and one in the blocks. However, if the element is present in both the log and the blocks, the copy in the log must have been the one that arrived later and is therefore the one returned during queries.

Operations

Concurrency control. As described in Chapter 2, the optimistic concurrency control mechanism in the BP-tree uses exclusive locks on each leaf. Therefore, the operations

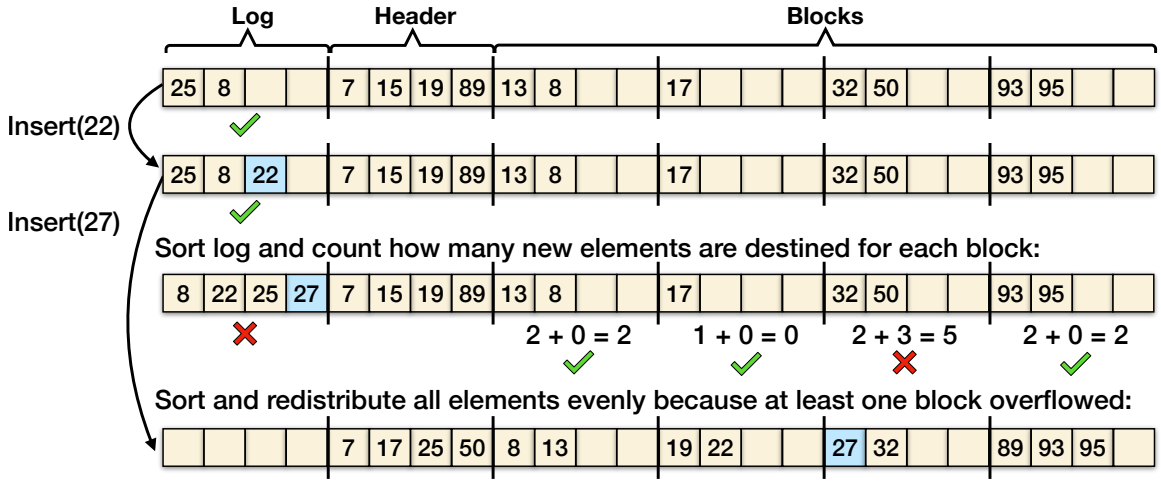

Figure 4-1: Examples of insertions in a BPA. For simplicity, we illustrate only the keys for each element. In this example, `log_size` = `num_blocks` = `block_size` = 4. Upon the first insertion (of 22), there is enough space in the log to hold the new element, so the element is placed at the end of the log. The second insertion (of 27) overflows the log, so the elements from the log are sorted and the BPA determines how many are destined for each block. Since flushing the log would overflow at least one of the blocks, the BPA sorts all blocks and redistributes elements evenly amongst them.

on the BPA do not need to be thread safe because the leaf node will be locked when performing the operation in the BP-tree.

Insert. The BPA is an insert-optimized data structure that supports fast inserts by buffering inserts and storing elements in a contiguous array partitioned into fixed-size blocks with empty spaces. The BPA maintains the invariant that there is always at least one empty cell in the log and in each block after an insertion has been completed.

Suppose we want to insert a key-value pair (`k, v`) into the BPA. The BPA first scans all elements in the log. If any of the elements in the log have `k` as their key, the BPA replaces that element with (`k, v`) and returns. Otherwise, it appends the element to the end of the log. There are two cases after adding the new element to the log:

Case 1: There is at least one empty cell left in the log.

The insertion is complete. The first insert in Figure 4-1 illustrates appending an element at the end of the buffer.

Case 2: There are no empty cells left in the log.

To maintain the invariant that there are empty cells in the log before any insert, the BPA sorts and then ***flushes***, or moves, the elements to the rest of the data structure based

on the partitioning given in the header. If this is the first time elements are being flushed from the log (e.g., near the beginning of the lifetime of the BPA) and there are no elements yet in the header, the log is ordered and simply moved to the header. Otherwise, if there are elements in the header, the BPA first counts up how many elements are in each block and stores the result in an array called `count_per_block` that stores how many elements are currently in each block. It then determines how many new elements (excluding duplicates) are destined for each block (based on the partitioning from the header) and stores the result in an array called `new_destined_per_block`. There are two possible cases:

Case 2a: A flush would not result in any block becoming full.

Formally, for any $i = 0,1,...,$`num_blocks`-1, we have `count_per_block[i] + new_destined_per_block[i]` $<$ `block_size`. In this case, each block has enough space to accomodate elements from the log while still maintaining the invariant that there is at least one empty space in each block. The BPA flushes elements from the log to a block in two steps. It first replaces any elements in the header/block with the same key as an element in the log with the newer version from the log. It then moves all other elements in the log destined for the block into that block. If there are currently elements in the block, the BPA appends any relevant elements from the log to the end of those elements. After the flush, the BPA completes the insertion by clearing the log.

Case 2b: A flush would result in at least one block becoming full.

Formally, there exists some $i = 0,1,...,$`num_blocks`-1 such that `count_per_block[i] + new_destined_per_block[i]` $\geq$ `block_size`. The second insert in Figure 4-1 illustrates this case of possibly filling one of the blocks upon a flush.

If there is not enough space in at least one of the blocks to flush elements from the log, the BPA sorts each block and merges all elements (from the log, header, and blocks) into a separate array, removing duplicate keys (i.e., if there are elements with the same key in the log and the header/blocks) along the way. At the end of the merge, all elements in the data structure are stored in sorted order in the separate array. The BPA then performs a ***redistribute***[1] that chooses a new header that split elements as evenly as

[1]The redistribute procedure is inspired by the Packed Memory Array (PMA) data structure [30, 7], but the PMA is distinct in that it may perform local redistributes that do not include all of the blocks, while the BPA only performs global redistributes of all blocks.

possible amongst the blocks.

After the redistribute, the BPA completes the insertion by clearing the log.

Delete. Just as in inserts, deletes in a BPA are buffered in the log and flushed to the header/blocks when there are no empty cells left in the log. Inserts and deletes use the same log, but inserts grow from the front of the log and deletes grow from the back. The BPA keeps track of how many inserts and deletes there are to determine which messages in the log are inserts/deletes.

Suppose we want to delete an element with the key `k` from the BPA. First, we scan the deletes that are currently in the log. If a delete for key `k` already exists, we exit because the delete will already be propagated to the rest of the BPA. If the delete did not already exist in the log, we scan the buffered inserts and remove any that have `k` as their key. Regardless of whether a matching element was found in the log, the BPA prepends a delete message with the key `k` (along with a filler null value if in map mode) to the end of the log. If the log is full, the BPA first flushes deletes to the header/blocks by deleting any elements with keys that match the delete messages. If the deletes affects the header or cause any of the blocks to become empty, the BPA performs a redistribute like in the insert case. If there was not a redistribute, the BPA then flushes the inserts.

Find. Point queries in a BPA first check the log, then the header, and then finally the blocks. If the key is found in the inserts in the log, the BPA returns that element. If the key is found in the deletes in the log, the BPA returns `null`. If the key is not found in the log, the BPA checks the header to see if the element is in the header. If the element is in the header, the BPA returns that element. If it is not in the header, the BPA uses the header to determine the block that the target element might possibly reside in. Finally, the BPA checks all elements in the relevant block and returns the element with the matching key, if there is one. Otherwise, it returns `null` because there was no element with a matching key.

Range iteration. Although the BPA is not globally sorted, it supports sorted iteration in a range (`iterate_range`) by sorting elements as necessary and then processing the elements in sorted order. Suppose the user calls the procedure `iterate_range(start, length, f)`. As a preprocessing step, the BPA first flushes all deletes in the log to the

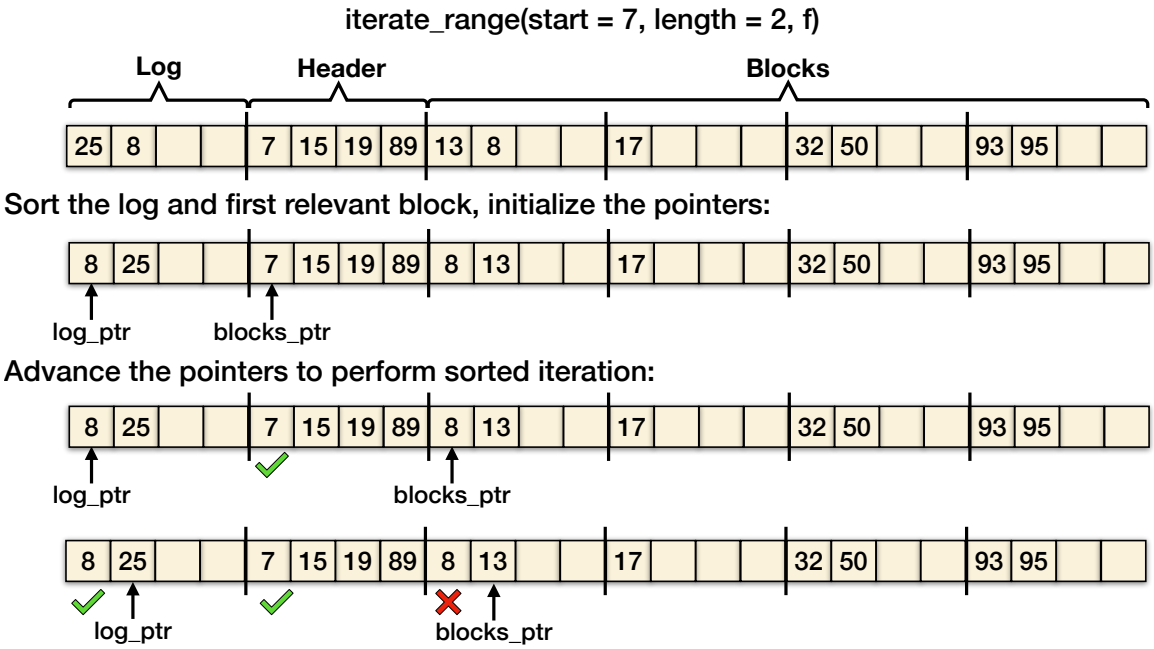

Figure 4-2: An example of a call to `iterate_range` in BPA. In this example, `log_size` = `num_blocks` = `block_size` = 4. The BPA sorts parts of the data structure as needed. To perform the ranged iteration, the BPA executes a two-finger merge through the relevant (and now sorted) parts of the BPA.

header/blocks. The BPA then sorts the remaining inserts in the log and the block that the `start` key would reside in. It then initializes two pointers:

- `log_ptr`: Initially points to the smallest element in the log that is greater than or equal to `start`.
- `blocks_ptr`: Initially points to the smallest element in the header/blocks that is greater than or equal to `start`.

The BPA then performs a co-iteration with the two pointers to process `length` elements and applies the function `f` to those elements while advancing the pointers as necessary. If the query is not finished but the `blocks_ptr` reaches the end of its current block, the BPA sorts the next block (if there is one) and moves the `blocks_ptr` to the start of that block. If either of the pointers reaches the end of their respective sections (the log or the blocks) and fewer than `length` elements have been processed, the BPA advances the remaining pointer until it reaches the correct number of elements or runs out of elements.

As an additional optimization to avoid unnecessary sorting, the BPA keeps a bit vector of length `num_blocks` that denotes whether the elements in each block are currently

sorted. It sorts a block during a range query if and only if the corresponding bit in the bit vector is unset. If the BPA sorts a block as part of a redistribute or sorted range query, it sets the corresponding bit. If there are elements flushed to a block during an insert, the BPA unsets the corresponding bit because the elements in the block may have become unsorted.

Figure 4-2 presents a worked example of a sorted range query.

Range map. The BPA also supports range maps queries of the form `map_range(start, end, f)`. The procedure for range maps is similar to the one for range iteration except that maps do not need to sort elements and perform a two-finger co-iteration. As a preprocessing step, the BPA first flushes all deletes in the log to the header/blocks. The map then scans through the inserts in the log and applies the function `f` to all elements that fall within the range. During this scan, the BPA searches for each of the elements in the log in the blocks to keep track of duplicates. After scanning through the log, the BPA scans through all blocks with elements that may fall into the range. If an element in the blocks has a newer version in the log, we skip the one in the blocks because it has already been accounted for. Otherwise, we apply the function `f` to any element in the blocks that falls into the range.

Chapter 5

The BP-tree

This chapter introduces the ***BP-tree***, a modified version of the concurrent B$^+$-tree described in Chapter 2 that used large leaf nodes to optimize range operation without slowing down point operations. It uses the BPA from Chapter 4 in the leaf nodes and standard sorted arrays in the internal nodes. As shown in Chapter 3, it is not necessary to replace the sorted array in every node with a BPA because all data records are in the leaf nodes and relatively few are in the internal nodes. Next, this chapter describes how to implement the B$^+$-tree operations outlined in Chapter 2 in the BP-tree. Finally, this chapter demonstrates the benefits of the BP-tree on range operations in the affine model.

Structure

The BP-tree replaces the sorted array in the leaves of a B$^+$-tree with the BPA. The leaf nodes in the BP-tree do not have to be the same size as the internal nodes, and often are much larger because of the specialized BPA data structure.

Traditional B$^+$-trees keep track of exactly how many elements are in each leaf to determine when a leaf becomes too full during insertions. Since the buffered partitioned array may contain duplicates of elements due to buffering, each leaf in a BP-tree keeps track of its `num_elts`, or the number of elements in each leaf (including duplicates), to determine how full the leaf node is. Although the count `num_elts` in the BP-tree may be an overestimate of the number of elements with different keys, it is at most `log_size`

away from the correct count.

Operations

Insert. Like inserts in the plain B^+-tree, inserts in the BP-tree first traverse down to the correct leaf by following the root-to-leaf path and then check if the target leaf is full (i.e., the count `num_elts` is equal to the number of slots in the BPA).

If the leaf is not full, we call `insert` on the BPA and increment the `num_elts` in the leaf. If the leaf is full, we first flush the log in the BPA in that leaf and then perform a split. A split creates a new "right" leaf and divides the elements as evenly as possible between the current leaf and the new leaf. Since `log_size` is much smaller than the size of the BPA, there are always enough elements in the BPA to perform a valid split even if all of the elements in the log are duplicates. The split moves the upper half of the full BPA's elements in sorted order into a new BPA structure. After a split, the current leaf contains the first half of elements in sorted order, and the new right leaf contains the new BPA with the remaining half of the elements. The BP-tree then checks which leaf the new element should be inserted into, and calls insert on that BPA.

Find. Finds in the BP-tree first traverse down to the leaf where the key would be located, then use the BPA's find functionality.

Range iteration. BP-tree range iterations use the `iterate_range` functionality in the BPA to process elements in sorted order at the leaves. Given a call to `iterate_range(start, length, f)` in the BP-tree, the first step is to traverse down to the leaf where the `start` key would be located. The BP-tree then calls the `iterate_range` on the BPA (as described in Chapter 4) with the same parameters. The BPA reports the number of elements found in the query. If the reported number of elements found equals the length of the query, the query is finished. Otherwise, the BP-tree keeps track of how many elements have been processed so far. It then continues onto the next leaf and calls `iterate_range` on the BPA in this new leaf with the remaining number of elements in the query and adjusts the count of elements

processed so far accordingly. This process of traversing leaves repeats until the total number of elements found is equal to `length`, or no next leaf exists.

The concurrency mechanism described in Chapter 2 states that we always grab exclusive locks on the leaf nodes including during find or range queries. Therefore, during the range iteration operation in BP-tree we can safely modify the BPA as described in Chapter 4 without modifying the concurrency mechanism in the BP-tree.

Range map. Range maps in a B-tree of the form `map_range(start, end, f)` traverse down to the leaf where the `start` key would be located, then use the `map_range` functionality in the BPA as described in Chapter 4 to apply the function `f` to all elements in the range. We then check if the maximum key in the leaf is greater than the `end` key, and if so, we are finished. Otherwise, we continue to traverse to the next leaf and repeat the range map until the the maximum key in the current leaf is greater than the `end` key, or no right leaf exists.

Analyzing range operations in the affine model. The affine model [6] captures the benefits of the BP-tree for range operations by modeling the cost of random vs sequential memory accesses. A range operation (iteration and map) consists of a search through the internal nodes of the tree and then a scan at the leaves. Since the changes in the BP-tree only affect the size of the leaves and not the higher-level search, we focus the analysis on the scan. Suppose that a range query performs a scan of size k. Given a leaf size of L, the scan at the leaves has cost

$$\Theta\left(\left(\frac{k}{L}\right)(1+\alpha L)\right)=\Theta\left(\frac{k}{L}+k\alpha\right)$$

in the affine model.

Let L_1 be the size of the leaves in a B^+-tree and L_2 be the size of leaves in a BP-tree. Since the BP-tree is optimized for large leaves, we have $L_1 < L_2$. When the length of the scan k is sufficiently large ($k=\Omega(L_1+L_2)$),

$$\frac{k}{L_1}+k\alpha=\Omega\left(\frac{k}{L_2}+k\alpha\right).$$

Chapter 6

Evaluation

This section demonstrates that the BP-tree improves long-running range operations without giving up point-operation performance on a suite of microbenchmarks as well as on workloads generated from YCSB [16]. The BP-tree supports range iteration up to 1.4× faster and range maps up to 1.7× faster when compared to the best-case insertion configuration for the B$^+$-tree. Furthermore, the BP-tree achieves slightly faster performance (about 1.1×) on point-operation workloads compared to the B$^+$-tree. The BP-tree achieves slower performance (within 1.15×) on the short-running range-operation workload compared to the B$^+$-tree, as the range fits within the leaf node and thus does not benefit as much from the locality in the larger leaf nodes in the BP-tree. To illustrate the use case for the BP-tree, we generated workloads with long-running range operations from YCSB and found that the BP-tree achieves between 1.2×-1.4× speedup compared to the B$^+$-tree.

Tables 6.4, 6.5, and 6.6 contain all data used to generate the plots in this section.

Systems setup

All experiments were run on a 48-core 2-way hyperthreaded Intel Xeon Platinum 8275CL CPU @ 3.00GHz with 189 GB of memory from AWS [2]. The machine has 1.5MiB of L1 cache, 48 MiB of L2 cache, and 71.5 MiB of L3 cache across all of the cores. To avoid non-uniform memory access (NUMA) issues across sockets, we ran all experiments on

a single socket with 24 physical cores and 48 hyperthreads. All times are the median of 5 trials after one warm-up trial.

Data structures setup

We used the B$^+$-tree [14] from the TLX library [8] with 64-bit elements in map mode (i.e., with keys and values) as the starting point for our implementation. We then implemented the optimistic concurrency control scheme described in Chapter 2 on top of the main operations. We ran operations concurrently using the OpenCilk compiler [42]. Finally, we implemented the BPA from Chapter 4 in the leaves of the TLX B$^+$-tree.

We tested various node sizes in two different types of blocked trees:

- The standard B$^+$-tree which sets the internal and leaf node sizes to be the same.
- The ***BP-tree*** with BPAs in the leaf nodes.

In Chapter 3, we tried a variant of B$^+$-trees that only grows the leaf nodes and keeps the size of the internal nodes fixed, but found that the performance was similar to the traditional B$^+$-tree because the leaf nodes are the most affected during operations.

The B$^+$-tree takes a parameter `node_size` (in bytes) for both the internal and leaf nodes. We tested different (powers of 2) node sizes ranging from 2^8 up to 2^{16}. We also tested non-powers of 2 node sizes ranging from 2^8 up to 2^{16} and found no discrepancies between node sizes of powers and non-powers of 2.

The BP-tree takes several parameters as described in Chapter 4: `internal_size`, `header_size`, and `block_size`. We tested two configurations of the BPA in the leaves to explore the effect of the number of blocks on the tree's performance. The ***small*** version sets `internal_size` = 1024 (bytes), `header_size` = 32 (slots), and `block_size` = 32 (slots) (for a total of 1088 slots per leaf). The ***large*** version sets `internal_size` = 1024 (bytes), `header_size` = 64 (slots), and `block_size` = 32 (slots) (for a total of 2176 slots per leaf). In both configurations, we set each block to take up 8 cache lines (at 64 bytes per line). Furthermore, we set each internal node to 1024 bytes but found that the size of the internal node did not have much of an effect.

Evaluation on microbenchmarks

Workloads setup. We concurrently inserted 100M uniform random elements in the range $[1, 2^{64}-1]$. We then performed finds (point queries) for 1M of those elements. Finally, we tested range queries with varying maximum lengths. For each maximum length `max_len` tested, we performed 1M range iterations with lengths distributed uniformly randomly in the interval [0, `max_len`]. We saved the start and end points of each of these queries and used them to perform 1M range maps on the same ranges. In addition to the total operation times shown in Figure 1-2, we also report individual operation latencies by percentile in Figure 6-1.

Inserts. Although the BP-tree has large leaf nodes (over 16k bytes), it achieves high insertion throughput because the optimized BPA data structure amortizes element moves across inserts. In the traditional B$^+$-tree, each insertion shifts existing elements within a leaf's sorted array to make space for the new element. Therefore, the insertion performance in B$^+$-trees with sorted arrays in the leaves degrades with increasing leaf size because the number of element moves grows proportionally with the leaf size. In contrast, the BPA relaxes the sortedness of the leaves and buffers insertions in the log. It flushes elements to the blocks only when the log is full, amortizing accesses to the blocks across inserts.

The BP-tree achieves similar (within 1.1×) insertion throughput when compared to the best-case insertion throughput of the baseline B$^+$-tree (at `node_size` = 1024). However, as shown in Table 6.4, the BP-tree is over 5× faster for inserts when compared to a B$^+$-tree with similar-sized leaf nodes. Figure 1-1 illustrates the decline in insertion throughput in the B$^+$-tree as the leaf size grows.

In addition to timing overall concurrent performance, we also timed each individual operation to analyze worst-case latency. Figure 6-1 shows the normalized individual operation speedup of the BP-tree relative to the B$^+$-tree at various percentiles (with higher percentiles indicating slower individual operation latency). For worst-case inserts, the 99.9th percentile of inserts in the BP-tree are approximately 50% slower than the 99.9th percentile of inserts in the B$^+$-tree. The slowdown in the worst-case is due to

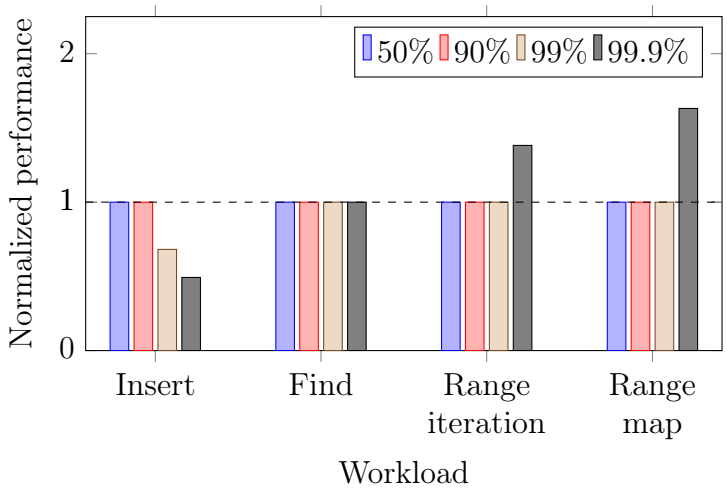

Figure 6-1: Normalized individual operation speedup of the BP-tree relative to the baseline B$^+$-tree by percentile. The x-axis lists the evaluated workloads. The y-axis represents the average latency of an individual operation in the B$^+$-tree relative to the average latency of an individual operation in the BP-tree (i.e. above 1 means the BP-tree is faster than the B$^+$-tree).

the amortization of insertions in the buffered partitioned array, where some inserts may cause a buffered partitioned array to flush or redistribute.

Finds. Figure 1-2 demonstrates that the BP-tree supports finds about 1.2× faster than the best-case B$^+$-tree configuration for finds (at `node_size` = 1024). Finding an element in a BPA avoids looking at the entire data structure via the header, which enables a lookup to skip to the relevant block that might contain the target element. In contrast, finding an element in an array requires a scan when the node size is small (up to 2048 bytes) or a binary search when the node size is large.

As shown in Figure 1-1, the find throughput does not change as much as the insert throughput as a function of node size in a B$^+$-tree. Insertion performance degrades more dramatically with larger node sizes because the number of elements that must be shifted upon an insert grows linearly with the size of the nodes. In contrast, when the nodes are sufficiently large, finds can be implemented in the B$^+$-tree with a binary search, which only requires looking at a logarithmic number of elements in each node. Figure 6-1 shows that individual operation performance for finds are equivalent between the BP-tree and B$^+$-tree across percentiles.

Range operations. As mentioned in Chapters 1 and 2, we evaluate two types of range operations: range iteration, which processes elements in sorted order (according to their

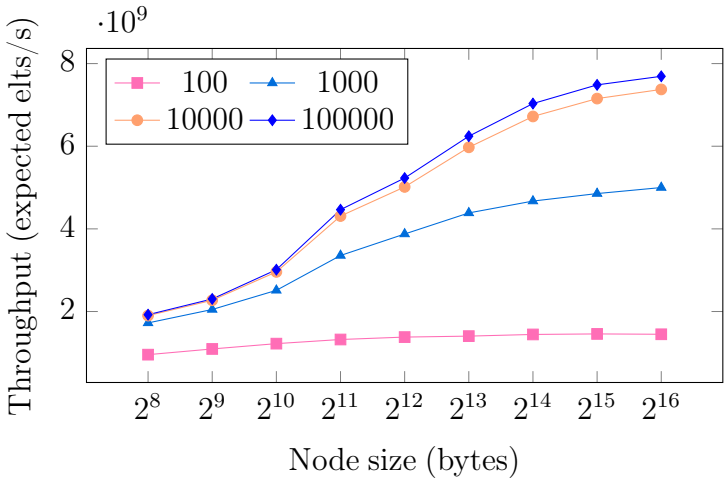

Figure 6-2: Throughput of range iterations of varying maximum lengths as a function of node size in B$^+$-trees.

keys) and range map, which processes elements in a range in any order.

Figure 6-2 demonstrates that the B$^+$-tree range iteration throughput (in terms of expected elements processed per second) improves with both the average range size and B$^+$-tree node size. As predicted by the discussion of the affine model in Chapter 5, longer range operations take advantage of locality more than short-running range operations because longer ranges process more contiguous elements. The shortest range operation (with maximum size 100) does not improve much with the node size because the queries are contained in one node. In contrast, the longest range operation (with maximum size 100,000) improves by about 3× as the node size grows. In traditional B$^+$-trees, both types of range operations have similar throughput because the arrays in the leaves are sorted, so we only plot the results for range iteration but report results for both types of range operations in Tables 6.5 and 6.6. Similarly, Figure 6-1 demonstrates that both worst-case individual range operations are improved in the BP-tree compared to the B$^+$-tree, as the slowest individual operations span longer ranges and thus benefit most from the increased leaf node size in the BP-tree.

Figure 6-3 reports the throughput of range operations of varying lengths in the best-case B$^+$-tree for inserts and the BP-tree (with the small BPA configuration). We find that the BP-tree is between $1.1-1.2\times$ slower for short-running range operations when compared to the B$^+$-tree because the BPA incurs computational overhead for its improved locality, but short-running ranges fit within the B$^+$-tree nodes and do not

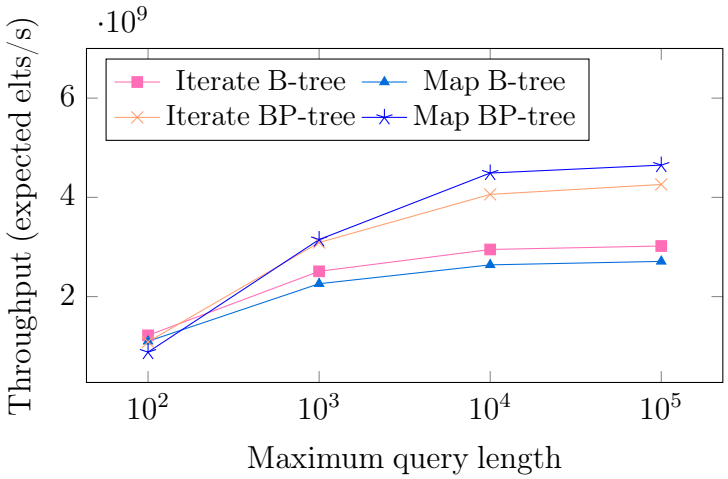

Figure 6-3: Range operation thorughput as a function of the maximum length of each query.

Baseline B+-tree		*Insert*		*Find*		*Range iteration*		*Range map*	
Internal size (bytes)	*Leaf size (bytes)*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*
256	256	1.10E7	0.39	3.87E7	0.92	1.93E9	0.25	1.84E9	0.24
512	512	2.24E7	0.79	4.08E7	0.96	2.30E9	0.30	2.12E9	0.28
1024	1024	2.82E7	1.00	4.23E7	1.00	3.02E9	0.39	2.71E9	0.35
2048	2048	2.35E7	0.83	4.21E7	1.00	4.47E9	0.58	4.24E9	0.55
4096	4096	1.58E7	0.56	4.08E7	0.97	5.23E9	0.68	5.14E9	0.67
8192	8192	9.23E6	0.33	4.04E7	0.96	6.24E9	0.81	5.84E9	0.76
16384	16384	4.96E6	0.18	4.05E7	0.96	7.03E9	0.91	6.57E9	0.86
32768	32768	2.58E6	0.09	4.02E7	0.95	7.48E9	0.97	7.25E9	0.95
65536	65536	1.31E6	0.05	3.92E7	0.93	7.69E9	1.00	7.67E9	1.00
BP-tree		*Insert*		*Find*		*Range iteration*		*Range map*	
Internal size (bytes)	*Leaf size (slots)*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1024	1088	3.25E7	1.15	5.33E7	1.26	4.26E9	1.41	4.65E9	1.72
1024	2176	3.00E7	1.06	4.91E7	1.16	4.55E9	1.51	4.86E9	1.79

Table 6.4: Throughput and normalized performance of point and range operations. Point operation throughput is reported in operations/s and range query throughput is reported in (expected elements processed)/s. We use N.P. to denote the normalized performance in the B$^+$-tree compared to the best B$^+$-tree configuration for that operation (1.0 is the best possible value). We use SU in the BP-tree to denote the speedup for each operation compared to the B$^+$-tree with `node_size` = 1024. The range operations are reported for the largest tested range (max length of 100,000).

benefit much from this better locality. However, the BP-tree achieves up 1.4× speedup on range iterations and 1.7× speedup on range maps compared to the B$^+$-tree on large ranges (i.e., when the length of the range operation grows larger than the B$^+$-tree node size). Iteration is slower in the BP-tree compared to maps because the BPA is not a sorted data structure. There is additional computational overhead to returning elements in sorted order in the BPA because the blocks and log must be sorted to perform the sorted scan. However, as mentioned in Chapter 1, many applications may not require sortedness in their range operations.

Baseline B+-tree		*max_len = 100*		*max_len = 1,000*		*max_len = 10,000*		max_len = *100,000*	
Internal size (bytes)	*Leaf size (bytes)*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*
256	256	9.56E8	0.66	1.73E9	0.35	1.90E9	0.26	1.93E9	0.25
512	512	1.10E9	0.75	2.05E9	0.41	2.27E9	0.31	2.30E9	0.30
1024	1024	1.22E9	0.84	2.51E9	0.50	2.95E9	0.40	3.02E9	0.39
2048	2048	1.32E9	0.91	3.36E9	0.67	4.31E9	0.58	4.47E9	0.58
4096	4096	1.38E9	0.95	3.88E9	0.78	5.02E9	0.68	5.23E9	0.68
8192	8192	1.41E9	0.96	4.38E9	0.88	5.97E9	0.81	6.24E9	0.81
16384	16384	1.44E9	0.99	4.69E9	0.94	6.72E9	0.91	7.03E9	0.91
32768	32768	1.46E9	1.00	4.87E9	0.98	7.15E9	0.97	7.48E9	0.97
65536	65536	1.45E9	0.99	4.99E9	1.00	7.37E9	1.00	7.69E9	1.00
BP-tree		*max_len = 100*		*max_len = 1,000*		*max_len = 10,000*		max_len = *100,000*	
Internal size (bytes)	*Leaf size (slots)*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1024	1088	1.09E9	0.89	3.09E9	1.23	4.06E9	1.37	4.26E9	1.41
1024	2176	9.34E8	0.76	3.07E9	1.22	4.31E9	1.46	4.55E9	1.51

Table 6.5: Throughput (in expected elements processed per second) of range iterations of varying maximum lengths (`max_len`) and normalized performance compared to the best-case performance for each maximum length (1.0 is the best possible value). We use SU in the BP-tree to denote the speedup for each operation compared to the B$^+$-tree with `node_size` = 1024.

Baseline B+-tree		*max_len = 100*		*max_len = 1,000*		*max_len = 10,000*		max_len = *100,000*	
Internal size (bytes)	*Leaf size (bytes)*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*	*Throughput*	*N.P.*
256	256	9.01E8	0.76	1.63E9	0.37	1.81E9	0.25	1.84E9	0.24
512	512	1.01E9	0.85	1.89E9	0.43	2.09E9	0.29	2.12E9	0.28
1024	1024	1.10E9	0.93	2.26E9	0.51	2.64E9	0.36	2.71E9	0.35
2048	2048	1.17E9	0.98	3.11E9	0.70	4.06E9	0.56	4.24E9	0.55
4096	4096	1.19E9	1.00	3.64E9	0.82	4.89E9	0.68	5.14E9	0.67
8192	8192	1.19E9	1.00	4.01E9	0.91	5.57E9	0.77	5.84E9	0.76
16384	16384	1.19E9	1.00	4.23E9	0.96	6.23E9	0.86	6.57E9	0.86
32768	32768	1.19E9	1.00	4.37E9	0.99	6.87E9	0.95	7.25E9	0.95
65536	65536	1.18E9	0.99	4.42E9	1.00	7.24E9	1.00	7.67E9	1.00
BP-tree		*max_len = 100*		*max_len = 1,000*		*max_len = 10,000*		max_len = *100,000*	
Internal size (bytes)	*Leaf size (slots)*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*	*Throughput*	*SU*
1024	1088	8.82E8	0.80	3.15E9	1.40	4.49E9	1.70	4.65E9	1.72
1024	2176	7.63E8	0.69	3.12E9	1.38	4.61E9	1.74	4.86E9	1.80

Table 6.6: Throughput (in expected elements processed per second) of range maps of varying maximum lengths (`max_len`) and normalized performance compared to the best-case performance for each maximum length (1.0 is the best possible value). We use SU in the BP-tree to denote the speedup for each operation compared to the B$^+$-tree with `node_size` = 1024.

Evaluation on YCSB workloads

We also evaluate the B$^+$-tree and BP-tree on workloads from YCSB [64] and report the results in Figure 1-4 and Table 6.7.

Experimental setup. Based on the results from Section 6, we fixed the parameters for the B$^+$-tree at `node_size` = 1024 bytes to maximize insertion throughput. For the BP-tree, we set the internal node size `internal_size` = 1024 (bytes) and used the small version of the BPA (with 1088 elements each) in the leaves.

Table 6.7 presents details of the core workloads from YCSB [64]. We tested work-

Workload	*Description*	*B^+-tree*	*BP-tree*	*BP-tree/B^+-tree*
Load	100% inserts	2.76E7	3.12E7	1.13
A	50% finds, 50% inserts	3.22E7	3.43E7	1.07
B	95% finds, 5% inserts	4.01E7	4.42E7	1.10
C	100% finds	4.23E7	4.61E7	1.09
E	95% short-running range iterations (max length 100), 5% inserts	2.75E7	2.30E7	0.84
X	100% long-running range iterations (max length 10,000)	6.82E5	8.21E5	1.20
Y	100% long-running range maps (max length 10,000)	6.34E5	9.21E5	1.45

Table 6.7: Throughput (in operations/s) of the B^+-tree and BP-tree on workloads from YCSB

loads[1] A, B, C, and E from the core YCSB workloads by adapting the YCSB driver from RECIPE [34]. Running a workload in YCSB has two consecutive phases: 1) the *load* phase, which adds elements to the data structure, and 2) the *run* phase, which performs operations specified by the workload. For each workload, we generated 100M elements to insert in the load phase and 100M operations to perform in the run phase. All elements were generated uniformly at random to match the distribution in RECIPE [34]. We ran all 100M operations in each phase concurrently.

The YCSB experiments use the `insert` (put), `find` (get), and `iterate_range` (scan) operations defined in Chapter 2. To generate scan operations, the YCSB workload generator takes as input a `max_len` parameter and generates range iteration operations with lengths uniformly distributed in the range [0,`max_len`].

As mentioned in Chapter 1, the core YCSB workloads focus mostly on point operations and short-running range operations. To illustrate the strenghts of the BP-tree, we added two new workloads: 1) workload X, which performs long range iterations (with maximum length 10,000), and 2) workload Y, which performs long-running range maps (with maximum length 10,000). Although the core YCSB workloads do not originally include range maps, we include them in addition to the provided workloads to illustrate how the different systems perform on operations from other application areas.

Finds and inserts. The BP-tree is slightly faster (within about 1.1×) compared to the B^+-tree for loading elements as well as on the workloads containing point operations (workloads A, B, and C). The BP-tree is optimized for long-running range operations but is competitive with the best-case B^+-tree for point operations despite the much larger leaves in the BP-tree because of the insert-optimized BPA in the leaves.

[1]We omit workload D from YCSB because it benchmarks the read-latest operation, which is not the focus of this work.

Range operations. Just as in the microbenchmarks, the BP-tree is about .8× as fast as the B$^+$-tree on short-running range iterations (workload E) because the benefits of improved locality do not outweigh the added computational overhead in the BP-tree when the range size fits in one node.

Since the BP-tree is optimized for long-running range operations, we use the YCSB workload generator to create workloads X and Y, which perform long-running range iterations and maps, respectively. The BP-tree is about 1.2× faster on workload X and 1.4× faster on workload Y when compared to the B$^+$-tree. Although the BP-tree incurs computational overhead during range operations due to buffering, it is faster on the whole for long-running range operations that take advantage of its improved locality from large leaf nodes.

Chapter 7

Related work

This thesis focuses on resolving the point-range tradeoff in B$^+$-trees because of the B$^+$-tree's ubiquity in real-world database and storage systems. Due to the B$^+$-tree's popularity, there has been significant effort devoted to improving its cache utilization during queries. For example, the CSB+-tree [53] is a cache-sensitive search tree that achieves efficient updates by storing child nodes in contiguous memory areas. However, previous work showed that the CSB+-tree exhibits a similar tradeoff between point and range operations depending on the node size [26]. The results have shown that using node size larger than a cache line results in better range operation performance, but that insertion performance suffers with much larger nodes. There is no clear winner between the B$^+$-tree and CSB+-tree because the B$^+$-tree outperforms the CSB+-tree on update-intensive workloads. Furthermore, Masstree [38] is a trie variant of the B$^+$-tree that achieves high query throughput with cache optimizations. Future work includes integrating the BPA introduced in this thesis into these B$^+$-tree variants to improve their empirical point-range tradeoff.

The log-structured merge (LSM) tree [44] is another hierarchical structure used frequently in key-value stores such as LevelDB [36] and RocksDB [56]. At a high level, the LSM tree contains multiple levels of tree-like structures of increasing size. Each of the tree-like structures is tuned for the medium it is stored in (e.g., in memory or on disk). The smallest level of the LSM tree must support efficient insertions, and all of the levels must support fast point and range queries. The BPA has the potential to improve the smallest level of the LSM tree by improving both point and range operations.

The skip list [52] has recently gained popularity as a randomized alternative to B^+-trees. Since traditional skip lists (based on linked lists) exhibit poor cache utilization, previous work [21, 61, 63, 66] optimized skip lists for cache utilization by collocating some of the elements. These cache-friendly skip lists use a blocked structure to improve cache utilization. Therefore, they should exhibit a similar point-range operation tradeoff to B^+-trees because they must choose a node size. The BPA data structure could improve these blocked variants of skip lists by enabling fast point and range operations.

Chapter 8

Conclusion

In this thesis, I have presented the BP-tree as a solution to the tradeoff between point and range operations in B^+-trees by replacing the sorted array in traditional B^+-tree leaves with the novel buffered partitioned array. I believe that the BP-tree strikes an appropriate balance between point-operation and range-operation performance. Traditionally, a user could decide to make B^+-tree nodes smaller to achieve the best possible point operation performance, or make the nodes larger to improve range operations but sacrifice point operations. The BP-tree is an ideal candidate for emerging applications that serve blended workloads of range and point operations with the same data store. The BP-tree is optimized for long-running range operations because analytics-based applications rely heavily on range operations. At the same time, the BP-tree supports similar point-operation performance compared to the best-case B^+-tree.

I see two primary avenues of future work. First, broadening the scope of supported operations in the BP-tree will increase the robustness of this study. The BP-tree supports both range iteration and range map, which underlie operations like database joins and intersections. A merge operation that merges two BP-trees in sorted order would be a valuable extension to the supported range-operation workloads. Efficient merges could enable the BP-tree to function as the in-memory component of structures like LSM trees, which require merging to structures on disk.

Second, this thesis' higher-level technique of replacing sorted arrays in the nodes of B^+-trees can be applied to other blocked structures as well. I am particularly inter-

ested in other usages for the BPA. Example applications could include integration into B-skiplists [21], or into enterprise B^+-tree-based key-value stores such as MongoDB [40] or CouchDB [17].

Overall, I am excited by the progress we made in developing the BPA and BP-tree, and I anticipate further investigation into their applications will yield compelling results.

Bibliography

[1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[2] Amazon. Amazon web services. https://aws.amazon.com/.

[3] Matthew Andrews, Michael A Bender, and Lisa Zhang. New algorithms for disk scheduling. *Algorithmica*, 32(2):277–301, 2002.

[4] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.

[5] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[6] Michael A Bender, Alex Conway, Martín Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, et al. Small refinements to the DAM can have big consequences for data-structure design. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, 2019.

[7] Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[8] Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers, 2018. https://panthema.net/tlx.

[9] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 251–264, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554, 2003.

[11] Cassandra. https://cassandra.apache.org.

[12] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. Realtime data processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1087–1098, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Shimin Chen, Phillip B Gibbons, and Todd C Mowry. Improving index performance through prefetching. *ACM SIGMOD Record*, 30(2):235–246, 2001.

[14] Douglas Comer. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[15] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, 2020.

[16] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154, 2010.

[17] CouchDB. https://couchdb.apache.org/.

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[19] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.

[20] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards scalable real-time analytics: An architecture for scale-out of OLXP workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, Aug 2015.

[21] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010.

[22] Goetz Graefe. Write-optimized B-trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30*, pages 672–683, 2004.

[23] Goetz Graefe. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)*, 35(3):1–26, 2010.

[24] Jörg Hakenberg, Wei-Yi Cheng, Philippe Thomas, Ying-Chih Wang, Andrew V Uzilov, and Rong Chen. Integrating 400 million variants from 80,000 human samples with extensive annotations: towards a knowledge base to analyze disease cohorts. *BMC Bioinformatics*, 17(1):1–13, 2016.

[25] Rui Han, Zhen Jia, Wanling Gao, Xinhui Tian, and Lei Wang. Benchmarking big data systems: State-of-the-art and future directions. *arXiv preprint arXiv:1506.01494*, 2015.

[26] Richard A Hankins and Jignesh M Patel. Effect of node size on the performance of cache-conscious B+-trees. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 283–294, 2003.

[27] HBase. https://hbase.apache.org/.

[28] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at Facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9, 2014.

[29] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Newnes, 2020.

[30] Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In *ICALP*, pages 417–431, 1981.

[31] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. *ACM SIGARCH Computer Architecture News*, 38(3):314–325, 2010.

[32] Donald E. Knuth. *The Art of Computer Programming*, volume 1 of *Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 3rd edition, 1998. (book).

[33] H.T. Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[34] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, Ontario, Canada, October 2019.

[35] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *International Colloquium on Automata, Languages and Programming*, 1981.

[36] LevelDB. http://ccrma.stanford.edu/jos/bayes/bayes.html.

[37] Cheng Li, Yue Lu, Qiaozhu Mei, Dong Wang, and Sandeep Pandey. Click-through prediction for advertising in Twitter timeline. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1959–1968, 2015.

[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 183–196, 2012.

[39] Andreas Meier and Michael Kaufmann. NoSQL databases. In *SQL & NoSQL Databases*, pages 201–218. Springer, 2019.

[40] MongoDB. https://www.mongodb.com/.

[41] University of Alberta. Lecture notes in CMPUT 379: Operating system concepts, February 2014.

[42] OpenCilk. https://www.opencilk.org/.

[43] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, et al. The case for RAMCloud. *Communications of the ACM*, 54(7):121–130, 2011.

[44] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[45] Prashant Pandey, Yinjie Gao, and Carl Kingsford. VariantStore: an index for large-scale genomic variant search. *Genome Biology*, 22(1):1–25, 2021.

[46] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. Terrace: A hierarchical graph container for skewed dynamic graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1372–1385, 2021.

[47] Apostolos N Papadopoulos, Spyros Sioutas, Christos Zaroliagis, and Nikolaos Zacharatos. Efficient distributed range query processing in Apache spark. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 569–575. IEEE, 2019.

[48] Jure Petrovic. Using memcached for data distribution in industrial environment. In *Third International Conference on Systems (icons 2008)*, pages 368–372. IEEE, 2008.

[49] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. Fast scans on key-value stores. *Proceedings of the VLDB Endowment*, 10(11):1526–1537, 2017.

[50] Pouria Pirzadeh, Junichi Tatemura, Oliver Po, and Hakan Hacıgümüş. Performance evaluation of range queries in key value stores. *Journal of Grid Computing*, 10(1):109–132, 2012.

[51] PostgreSQL. https://www.postgresql.org/.

[52] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, Jun 1990.

[53] Jun Rao and Kenneth A. Ross. Making B+- trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, May 2000.

[54] Kai Ren and Garth Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 145–156, 2013.

[55] Vincent Reniers, Dimitri Van Landuyt, Ansar Rafique, and Wouter Joosen. On the state of NoSQL benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, page 107–112, New York, NY, USA, 2017. Association for Computing Machinery.

[56] RocksDB. http://rocksdb.org/.

[57] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[58] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[59] ScyllaDB. https://www.scylladb.com/.

[60] Julian Shun and Guy E Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146, 2013.

[61] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. Cache-sensitive skip list: Efficient range queries on modern CPUs. In *Data Management on New Hardware*, pages 1–17. Springer, 2016.

[62] Xiaohui Xie, Jun Lu, EJ Kulbokas, Todd R Golub, Vamsi Mootha, Kerstin Lindblad-Toh, Eric S Lander, and Manolis Kellis. Systematic discovery of regulatory motifs in human promoters and 3'UTRs by comparison of several mammals. *Nature*, 434(7031):338–345, 2005.

[63] Zhongle Xie, Qingchao Cai, HV Jagadish, Beng Chin Ooi, and Weng-Fai Wong. PI: a parallel in-memory skip list based index. *arXiv preprint arXiv:1601.00159*, 2016.

[64] YCSB. Core workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads, 2020.

[65] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.

[66] Jingtian Zhang, Sai Wu, Zeyuan Tan, Gang Chen, Zhushi Cheng, Wei Cao, Yusong Gao, and Xiaojie Feng. S3: A scalable in-memory skip-list index for key-value store. *Proceedings of the VLDB Endowment*, 12(12):2183–2194, 2019.