Thesis Proposal:
Parallel Balanced Binary Trees Using Just Join

Yihan Sun

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Guy E. Blelloch, Chair
Andrew Pavlo
Daniel D. K. Sleator
Michael T. Goodrich, University of California, Irvine

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Keywords: Maps, Key-value pairs, Augmented trees, Binary Search Trees, Parallel, Balancing Scheme

Abstract

Tree data structures play an important role in almost every area in computer science. Nowadays the explosion of data puts forward a higher demand for the performance of the trees, which requires efficient parallel algorithms for processing large-scale data, as well as a full-featured interface for achieving more functionalities.

Traditional algorithm designing on balanced BSTs based on insertions and deletions are plagued with challenges such as being hard to parallelize, single-operand oriented, and varying among different balancing schemes. This thesis proposes a parallel algorithmic framework that overcomes these challenges, which captures all balancing criteria in a single function JOIN. The JOIN-based algorithms are then used for supporting sequences, ordered sets, ordered maps and augmented maps (formally defined in this thesis). A wide variety of functions form a hierarchical interface for these four data types, which are all implemented by join-based algorithms as part of a library PAM (Parallel Augmented Maps). This library is parallel, work-efficient, generic across balancing schemes, persistent, safe for concurrency and applicable to a wide range of applications and queries by proper augmentations.

The augmented map structure itself is an abstraction befitting many real applications. The PAM interface greatly simplifies the programming of many applications over direct implementations, such as interval trees, range trees, inverted indices, segment trees, and database snapshot isolations.

Experiments show that the implementations of the functions in PAM, as well as all the applications using PAM are practically efficient. Sequentially the code achieves performance that matches or exceeds exiting libraries designed specially for a single application, and the parallel implementation gets speedups ranging from 40 to 90 on 72 cores with 2-way hyperthreading.

Contents

1 Introduction

One of the most important data structure used in algorithm design and programming is the *tree* structure. It is widely used to support other data types and applications such as organizing strings (e.g., tries [37, 45] and suffix trees [90]), hierarchical data (e.g., evolutionary trees and directory trees), sequences, ordered sets and maps, priority queues (e.g., heaps or search trees), databases and data warehouses (e.g., the log-structured merge-tree [75]), file systems, and many geometry algorithms (e.g., segment trees [14], range trees [15], kd-trees [15]). Designing efficient algorithms for trees has always been a diligent pursuit for researchers in different areas, and are essential for both bounding the theoretical costs and improving practical performance of these applications. In the modern world with very large data set, trees play an important role in efficiently managing, retrieving and analyzing big data, and are employed in many real-world database systems [35, 36, 57, 63, 64, 66, 82, 83]. Thus in modern programming, a tree structure should be able to process large amounts of data, as well as to efficiently handle very complicated operations. The former necessitates *parallelism* and/or *concurrency* in trees, and the later requires a full-featured interface for supporting multiple data types. This thesis proposes a large variety of parallel algorithms supporting an interface for four data types (see details later) using trees along with their efficient implementations.

In this thesis we limit ourselves to binary trees[1], which can be either empty (a LEAF) or some data u and with two children (noted as left and right) denoted as $\text{NODE}(T_L, u, T_R)$. Many operations on trees have cost proportional to the *height* of the tree. The height of a binary tree with n keys has a lower bound of $\log n$, and can degenerate to n in the worst case. Thus for performance, it is necessary to organize binary trees in a "near" balance manner. As a result, many *balancing schemes*[2], such as AVL trees [2], red-black trees [12], weight-balanced trees [73], treaps [86], splay trees [87], etc., are designed for bounding the height of trees. In general they all keep the tree height to be bounded in $O(\log n)$ (w.h.p. for treaps, amortized for splay trees).

Conventional algorithm design on balanced binary trees usually uses insertion and deletion as primitives for updates, but are plagued with some problems in parallel. First these two primitives are inherently sequential, especially when multiple updates occur at the same location. Secondly they both operate on a single element at a time, which is less practical for implementing bulk functions. Also, conventional algorithms on balanced trees are usually proposed for a specific balancing scheme, making implementations less generic even in the sequential setting, and much more complicated in parallel.

This thesis proposes the primitive JOIN [21, 89] for balanced binary trees, and a corresponding parallel algorithmic framework that can overcome these problems. The function $\text{JOIN}(T_L, e, T_R)$ for a given balancing scheme takes two balanced binary trees T_L, T_R balanced by that balancing scheme, and a single entry e as inputs, and returns a new valid balanced binary tree, that has the same entries and the same in-order traversal as $\text{NODE}(T_L, e, T_R)$, but satisfies the balancing criteria. One advantage of using

[1]We note that similar methodology can be extended more generally.

[2]Balancing schemes are often discussed in the context of binary search trees (BSTs), but are actually more general, i.e., the balancing criteria are independent of keys. For example, later in this thesis, balanced binary trees are used for supporting sequences. In that case the tree is not a search tree.

JOIN as the primitive is that it can easily enable the use of divide-and-conquer in algorithms (e.g., divide trees into parts, solve subproblems and JOIN them back as the result), which can be parallelized easily. JOIN is especially useful for some bulk operations such as merging two trees. Most importantly, JOIN captures and isolates all balancing criteria from the other functions, such that a variety of tree operations can be implemented generically without knowing the actual balancing scheme and are still efficient.

The JOIN-based algorithms support a variety of functions simply and effectively, which, step by step, form interfaces for four abstract data types: sequences, ordered sets, ordered maps and augmented maps (defined later in the thesis). This thesis introduces the algorithms, interfaces, implementations and applications of these four data types [3] using join-based algorithms on balanced trees.

1. **`Sequence(K)`**. A sequence of keys of type K can be maintained in a balanced binary tree. JOIN supports many generic functions independent of balancing schemes for sequences such as construction, append, split-at (split at the i-th element), etc.
2. **`Ordered_set(K,`**$<_K$**`)`**. On top of the sequence interface, if K have a total ordering, which associates K a comparison function $<_K : K \times K \mapsto$ **BOOL**, then an ordered *set* implementation can be formed. This also makes the tree a *binary search tree* (BST) from this point. Beyond the sequence functions, many aggregate functions for sets (e.g., UNION, INTERSECT, DIFFERENCE, etc.) are then added to the interface. Even the insertion and deletion can be done using JOIN within time asymptotically no more than the best-known sequential algorithms, i.e., $O(\log n)$.
3. **`Ordered_map(K,V,`**$<_K$**`)`**. On top of the set interface, if each key is also assigned a value of type V, then an interface for the ordered *map* (also known as key-value store, dictionary, table, or associative array) can be built. The functions in the set interface are then extended for accepting key-value pairs as entries.
4. **`Augmented_map(K,V,`**$<_K$**`,aug)`**. More functionalities of trees are usually achieved by smartly *augmenting* it. In this thesis the abstract data type supported by a special type of augmented trees is formally defined as the *augmented map*, which associates an augmentation structure aug to each map[4]. This augmentation structure are chosen based on the specific desired applications, and is consist of an augmented value type A, a base function $g : K \times V \mapsto A$, a combine function $f : A \times A \mapsto A$, and f's identity $a_\emptyset \in A$. Formal definitions and more details can be found in Chapter 5. The augmented map data type is specially designed for quick range-based operations. On top of the ordered map interface, some functions can be supported more efficiently making use of the augmentation, which forms an interface specific for augmented maps.

Based on this JOIN-based algorithmic framework, the interface of parallel sequences, ordered sets, maps and augmented maps is implemented in a library PAM (Parallel Augmented Maps), which is parallel, work-efficient, generic across balancing schemes, persistent and safe for concurrency.

The top-most level of the interface, which is the augmented map, is an abstraction befitting many real applications from simple quick range sum queries to inverted indices and many geometry applications such as range trees and segment trees. The augmented map abstraction and the PAM interface greatly simplify the implementation of these applications. Each single application mentioned in this thesis,

[3] They are somehow almost all datatypes one would like to support by binary trees.

[4] Note that as a data type, the augmented map abstraction itself does not depend on augmented trees. See Chapter 5 for more details.

which needs thousands of lines of code even sequentially, can be implemented atop the PAM library, in parallel, with around 100 lines of code.

Beyond being very concise and effective, the implementation based on PAM is also parallel and efficient both theoretically and in practice (See Table 8.1). Sequentially the code achieves performance that matches or exceeds exiting libraries designed specially for a single application, and the parallel implementation gets speedups ranging from 40 to 90 on 72 cores with 2-way hyperthreading.

Thesis Statement . *Using* JOIN *as a primitive for balance binary trees yields efficient parallel algorithms for sequences, ordered sets, ordered maps and augmented maps, that are simple, theoretically and practically efficient, balancing-scheme-independent and applicable to a wide range of applications.*

Proposed topics in the thesis. The proposed topics in this thesis include:

1. An algorithmic framework for the interface of sequences, sets, maps and augmented maps that builds all functions upon a single function JOIN. The framework is implemented using balanced binary trees, and is generic across four balancing schemes: AVL trees, red-black trees (RB trees), weight-balanced trees (WB trees) and treaps. Under this framework, (asymptotically) theoretically efficient parallel algorithms along with their bound proofs are given (partially done in [21]).
2. The formal definition and interface of the augmented map structure (partially done in [88]).
3. A parallel library (the Parallel Augmented Map library, PAM) that efficiently implements the functions on sequences, ordered sets, ordered maps and augmented maps using JOIN-based algorithms. Beyond parallelism and augmentation, this library also supports persistence and is safe for concurrency (partially done in [21, 88]).
4. A parallel sweepline paradigm using augmented tree structure as subroutine for computational geometry applications (partially done in [19]).
5. Some examples of use cases for augmented maps. The example applications include the interval tree, the 2D range tree, the 2D segment tree, sweepline algorithms for 2D range queries and segment queries, inverted indices, etc. (partially done in [19, 88])
6. An implementation for supporting snapshot isolation in databases. This is achieved by making use of the persistence property of the PAM library (future work).
7. An implementation for write-efficient operations on ordered sets and maps (future work).
8. Experimental results on all proposed algorithms in PAM, as well as all applications using PAM. They are compared with available existing libraries and algorithms and show very good performance (are competitive or outperform existing implementations) (partially done in [19, 21, 88]).

2 Related Work

Previous Work on Set Operations. Parallel set operations on two ordered sets have been well-studied. Paul, Vishkin, and Wagener studied bulk insertion and deletion on 2-3 trees in the PRAM model [78]. Park and Park showed similar results for red-black trees [77]. These algorithms are not based on JOIN and are not work efficient, requiring $O(m \log n)$ work. Katajainen [56] claimed an algorithm with $O\left(m \log(\frac{n}{m} + 1)\right)$ work and $O(\log n)$ span using 2-3 trees, but it appears to contain some bugs in the analysis [18]. Blelloch and Reid-Miller described a similar algorithm as Adams' (as well as ours) on treaps with optimal work (in expectation) and $O(\log n)$ span (with high probability) on a EREW PRAM with scan operations. This implies $O(\log n \log m)$ span on a plain EREW PRAM, and $O(\log n \log^* m)$ span on a plain CRCW PRAM. The pipelining that is used is quite complicated. Akhremtsev and Sanders [6] recently describe an algorithm for array-tree UNION based on (a, b)-trees with optimal work and $O(\log n)$ span on a CRCW PRAM. Our focus in this paper is in showing that very simple algorithms are work efficient and have polylogarithmic span, and less with optimizing the span.

Many researchers have studied concurrent algorithms and implementations of maps based on balanced search trees focusing on insertion, deletion and search [10, 17, 27, 29, 40, 43, 46, 60, 61, 65, 71]. Motivated by applications in data analysis recently researchers have considered mixing atomic range scanning with concurrent updates (insertion and deletion) [11, 28, 79]. None of these are work efficient for UNION since it is necessary to insert one tree into the other requiring at least $\Omega(m \log n)$ work. Also, none of them, has considered range sums taking sub-linear time.

Related Work on Fast Range Sums. There are many theoretical results on efficient sequential data-structures and algorithms for range-type queries using augmented trees in the context of specific applications such as interval queries, k-dimensional range sums, or segment intersection queries (see e.g. [69]). Several of these approaches have been implemented as part of systems [59, 72]. Our work is motivated by this work and our goal is to parallelize many of these ideas and put them in a framework in which it is much easier to develop efficient code. We know of no other general framework as described in Section 5.

Various forms of range queries have been considered in the context of relational databases [31, 47, 48, 49, 50]. Ho et. al. [50] specifically consider fast range sums. However the work is sequential, only applies to static data, and requires that the sum function has an inverse (e.g. works for addition, but not maximum). More generally, we do not believe that traditional (flat) relational databases are well suited for our approach since we use arbitrary data types for augmentation—e.g. our 2d range tree has augmented maps nested as their augmented values. Recently there has been interest in range queries in large clusters under systems such as Hadoop [3, 7]. Although these systems can extract ranges in work proportional to the number of elements in the range (or close to it), they do not support fast range sums. None of the "nosql" systems based on key-value stores [8, 67, 76, 84] support fast range sums.

3 Preliminaries

The notation in this thesis is summarized in Table 3.1, and will be introduced in the rest of this chapter.

Binary Search Trees and Balancing Schemes. A *binary tree* is either a LEAF, or a node consisting of a *left* binary tree T_L, some data u stored at the root, and a *right* binary tree T_R, denoted as $\text{NODE}(T_L, u, T_R)$. The *size* of a binary tree, or $|T|$, is 0 for a LEAF and $|T_L| + |T_R| + 1$ for a $\text{NODE}(T_L, u, T_R)$. The *weight* of a binary tree, or $w(T)$, is one more than its size (i.e., the number of leaves in the tree). The *height* of a binary tree, or $h(T)$, is 0 for a LEAF, and $\max(h(T_L), h(T_R)) + 1$ for a $\text{NODE}(T_L, u, T_R)$. *Parent*, *child*, *ancestor* and *descendant* are defined as usual (ancestor and descendant are inclusive of the node itself). The *left spine* of a binary tree is the path of nodes from the root to a leaf always following the left tree, and the *right spine* the path to a leaf following the right tree. The *in-order values* of a binary tree is the sequence of values returned by an in-order traversal of the tree.

A *balancing scheme* for binary trees is an invariant (or set of invariants) that is true for every node of a tree, and is for the purpose of keeping the tree nearly balanced. In this thesis four balancing schemes that ensure the height of every tree of size n is bounded by $O(\log n)$ are considered. For each balancing scheme a quantity *rank* of a tree is defined, denoted as $r(T)$.

AVL trees [2] have the invariant that for every $\text{NODE}(T_L, v, T_R)$, the height of T_L and T_R differ by at most one. This property implies that any AVL tree of size n has height at most $\log_\phi(n+1)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. For AVL trees $r(T) = h(T) - 1$.

Red-black (RB) trees [12] associate a color with every node and maintain two invariants: (the red rule) no red node has a red child, and (the black rule) the number of black nodes on every path from the root down to a leaf is equal. Unlike some other presentations, in this thesis the root of a tree is not required to be black. Our proof of the work bounds requires allowing a red root. The *black height* of a node T, denoted $\hat{h}(T)$, is defined to be the number of black nodes on a downward path from the node to a leaf (inclusive of the node). Any RB tree of size n has height at most $2\log_2(n+1)$. In RB trees $r(T) = 2(\hat{h}(T) - 1)$ if T is black and $r(T) = 2\hat{h}(T) - 1$ if T is red.

Weight-balanced (WB) trees with parameter α (also called BB$[\alpha]$ trees) [73] maintain for every $T = \text{NODE}(T_L, u, T_R)$ the invariant $\alpha \leq \frac{w(T_L)}{w(T)} \leq 1 - \alpha$. Two weight-balanced trees T_1 and T_2 have *like* weights if $\text{NODE}(T_1, u, T_2)$ is weight balanced. Any α weight-balanced tree of size n has height at most $\log_{\frac{1}{1-\alpha}} n$. For $\frac{2}{11} < \alpha \leq 1 - \frac{1}{\sqrt{2}}$ insertion and deletion can be implemented on weight balanced trees using just single and double rotations [24, 73]. We require the same condition for our implementation of JOIN, and in particular use $\alpha = 0.29$ in experiments. For WB trees $r(T) = \lceil \log_2(w(T)) \rceil - 1$.

Treaps [86] associate a uniformly random priority with every node and maintain the invariant that the priority at each node is no greater than the priority of its two children. Any treap of size n has height $O(\log n)$ with high probability (w.h.p). For treaps $r(T) = \lceil \log_2(w(T)) \rceil - 1$.

For all the four balancing schemes $r(T) = \Theta(\log(|T| + 1))$. In this thesis, with clear context, a tree node u is also used to denote the whole subtree T_u rooted at it, the entry stored in it, as well as the map that its subtree T_u represents, and vice versa.

A *Binary Search Tree* (BST) is a binary tree in which each value is a key taken from a total order,

Notation	Description
$root(T)$	The data at the root of T
T	The (sub)tree T or the root of T (with clear context)
u	The tree node u or the data stored at u (with clear context)
$\|T\|$ or $\textsc{Size}(T)$	The size of tree T
$h(T)$	The height of tree T
$\hat{h}(T)$	The black height of an RB tree T
$r(T)$	The rank of tree T
$w(T)$	The weight of tree T (i.e, $\|T\|+1$)
$p(T)$	The parent of node T
$k(u)$ or $k(e)$	The key stored tree node u or entry e
$v(u)$ or $v(e)$	The value stored tree node u or entry e
$e(u)$	The entry stored in tree node/data u
$\mathcal{A}(T)$ or $\mathcal{A}(u)$	The augmented value of tree T or stored at u
$C_L(T)$ or $C_L(u)$	The left subtree of tree T or the left child of node u
$C_R(T)$ or $C_R(u)$	The right subtree of tree T or the left child of node u
$\textsc{Expose}(T)$	$(C_L(T), root(T), C_R(T))$
$\textsc{Node}(L, e, R)$	A tree with entry e at root, left subtree L and right subtree R
$\textsc{NewNode}(e)$	Create a tree node with entry e

Table 3.1: Summary of notation.

and for which the in-order values are sorted. A *balanced BST* is a BST maintained with a balancing scheme, and is an efficient way to represent ordered sets.

Parallel Computational Model. Our algorithms are based on nested parallelism with nested fork-join constructs and no other synchronization or communication among parallel tasks.[1] All algorithms are deterministic. We use work W and span (or depth) S to analyze asymptotic costs, where the work is the total number of operations and span is the critical path. We use the simple composition rules $W(e_1 \parallel e_2) = W(e_1)+W(e_2)+1$ and $S(e_1 \parallel e_2) = \max(S(e_1), S(e_2))+1$. For sequential computation both work and span compose with addition. Any computation with W work and S span will run in time $T < \frac{W}{P} + S$ assuming a PRAM (random access shared memory) with P processors and a greedy scheduler [25, 26].

Sequences, Ordered Sets and Ordered Maps. This section reviews the definitions of sequences, ordered sets and ordered maps in mathematics and computer science.

For a universe of keys K, a sequence $s = \mathbb{SQ}(K)$ is a relation $R \subset [n] \times K$, where $[n]$ denotes the first n natural numbers[2]. The position of an element in a sequence is its *index*. A sequence $s = \mathbb{SQ}(K)$ of size n is noted as $\{s[1], s[2], \ldots, s[n]\}$ ($s_i \in K$) ordered by their indices. $s[i]$ is used to extract the i-th element in a sequence s. We note that typically the braces mean *sets*, which do not contain duplicates and do not require an order. Here for simplicity and consistency with extensions to ordered sets and maps, we use braces for sequences, as well as ordered sets, ordered maps and augmented maps. Inside

[1] This does not preclude using our algorithms in a concurrent setting.

[2] A sequence can certainly be of infinite length, where the domain is the natural number set $\mathbb{N}$.

the braces the elements are by default ordered by the corresponding criteria (i.e., indices for sequences, and the order on keys for ordered sets, ordered maps and augmented maps). This thesis also burrows some relations and functions such as $\subset$, $\in$, $\cup$, $\cap$ from sets.

For a universe of keys K, a set is a (well-defined) collection of distinct objects in K. An ordered set is a set for which K has a total order defined by $<_K$ (we drop the subscript in clear context). It is noted as $\mathbb{OS}(K, <_K)$. The ordered set can be maintained by a sequence $\mathbb{SQ}(K)$ that 1) there exists a comparison function $<_K : K \times K \mapsto \textbf{BOOL}$ that defines the total order on K, and 2) for any two valid indices i and $j > i$, $s[i] < s[j]$.

For a universe of keys K and values V, a *map* is a relation $R \subset K \times V$ (set of key-value pairs), such that a key can only appear in one element. An *ordered map* is a map for which the keys K have a total order defined by $<_K$, and is denoted as $\mathbb{OM}(K, <_K, V)$. The ordered set can be viewed as a special type of the ordered map with empty values.

The definition of the *augmented map*, as part of the contribution of this thesis, is given in Chapter 5.

In this thesis, the *entry* is defined on each of the four data types, and is denoted as e belongs to some entry type E. On sequences and ordered sets, each key is noted as an entry, and the entry type E is equivalent to the key type K. On ordered maps and augmented maps, each key-value pair (k, v) is viewed as an entry, and the entry type is $K \times V$. This thesis also uses the functions to extract the key and value from e as $k(e)$ and $v(e)$ (for sequences and sets $v(e)$ is invalid).

Persistent Data Structures. A persistent data structure [41] is a data structure that preserves the previous version of itself when it is modified and always yields a new updated structure. For tree structures persistence is often achieved by path copying, meaning that only the affected path related to the update is copied, such that the asymptotical cost remains unchanged. Nodes are largely shared across different versions of trees, making implementations space-efficient.

Snapshot Isolation of Database Systems. In databases, snapshot isolation [16] is a guarantee that all reads made in a transaction will see a consistent snapshot of the database (in practice it reads the last committed values that existed at the time it started), and the transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot. Snapshot Isolation is a multi-version (MV) method, so single-valued (SV) histories do not properly reflect the temporal operation sequences. At any time, each data item might have multiple versions, created by active and committed transactions. Reads by a transaction must choose the appropriate version.

Read-Write Asymmetry in NVM. The future of main memory appears to lie in a new wave of non-volatile memory (NVM) technologies (e.g., phase-change memory, spin-torque transfer magnetic RAM, memristor-based resistive RAM, conductive-bridging RAM) that promise persistence, significantly lower energy costs, and higher density than the DRAM technology used in today's main memories [51, 55, 70, 93]. A key property of such memory technologies, however, is their asymmetric read-write costs: writes can be an order of magnitude or more higher energy, higher latency, lower (per-module) bandwidth, and prone to wear-out than reads [5, 9, 20, 22, 30, 32, 38, 39, 52, 54, 58, 62, 68, 80, 91, 92, 94, 95]. This motivates the need for algorithms that are *write-efficient* in future database systems as well as general programming using the new memory, in that they significantly reduce the number of writes compared to existing algorithms. Blelloch et al. [13, 20, 23] first defined several sequential and parallel computation models that take asymmetric read-write costs into account. The models extend the well-known external-memory model [4] and contain a parameter ω, which corresponds to the cost of a write relative

to a read to the non-volatile memory. The model assumes a *small-memory* (cache) of size $M \geq 1$, and a *large-memory* of unbounded size. Both memories are organized in blocks (cache-lines) of B words. The CPU can only access the small-memory (with no cost). Transferring a block from large-memory to small-memory takes unit cost; on the other direction, the cost is either 0 if this block is clean and never modified, or $\omega \gg 1$ otherwise. Theoretical results on this new model have been studied by Blelloch et al. in [13, 20, 22].

Other Definitions. For an associative binary function f, we extend it as follows:

$$f(\square) = a_{\emptyset} \tag{3.1}$$

$$f(a_1) \equiv a_1 \tag{3.2}$$

$$f(a_1, a_2, \ldots, a_n) \equiv f(a_1, f(a_2, a_3, \ldots, a_n)) \tag{3.3}$$

Here $\square$ means an empty element.

We use the notation $[n]$ to indicate the range $[1, \ldots, n]$. For a sequence s, "$s[i..j]$" for $i < j$ means all entries in s from index i to index j in order.

We use "with high probability" (w.h.p.) to mean with probability at least $1 - n^{-c}$ for any constant $c > 0$, and with the constant in the big-O linear in c.

4 Join-based Algorithms for Sequences, Ordered Sets and Ordered Maps

This chapter introduces join-based algorithms on balanced BSTs. The JOIN function captures all balancing criteria such that it is the only function tackling with rebalancing and rotations. Making use of JOIN, all the other functions are generic in code. The algorithms for the JOIN function, designed specifically for each balancing scheme, are introduced in [21]. The property of the JOIN functions proposed in [21] is summarized in Theorem 4.0.1 (the *rank* of each balancing scheme is defined in Chapter 3).

Theorem 4.0.1 ([21]). *For AVL, RB and WB trees,* JOIN(T_L, k, T_R) *does* $O(|r(T_L) - r(T_R)|)$ *work. For treaps* JOIN *does* $O(\log t)$ *work in expectation if* k *has the* t*-th highest priority among all keys. For AVL, RB, WB trees and treaps,* JOIN *returns a tree* T *for which the rank satisfies* $\max(r(T_L), r(T_R)) \leq r(T) \leq 1 + \max(r(T_L), r(T_R))$.

FOR SEQUENCES:

SPLITLAST(T) =
 $(L, u, R) =$ EXPOSE(T);
 if $R =$ LEAF **then** (L, u)
 else let $(T', u') =$ SPLITLAST(R);
  in (JOIN$(L, e(u), T'), u'$)

JOIN2(T_L, T_R) =
 if $T_L =$ LEAF **then** T_R
 else let $(T'_L, u) =$ SPLITLAST(T_L);
  in JOIN$(T'_L, e(u), T_R)$

FOR ORDERED SETS:

SPLIT(T, k) =
 if $T =$ LEAF **then** (LEAF,□,LEAF)
 else $(L, u, R) =$ EXPOSE(T);
  if $k = k(u)$ **then** $(L, e(u), R)$
  else if $k < k(u)$ **then**
   $(L_L, b, L_R) =$ SPLIT(L, k);
   $(L_L, b,$ JOIN$(L_R, e(u), R))$
  else let $(R_L, b, R_R) =$ SPLIT(R, k);
   in (JOIN$(L, e(u), R_L), b, R_R)$

INSERT(T, e) =
 if $T = \emptyset$ **then** SINGLETON(e)
 else let $\langle L, u, R\rangle =$EXPOSE(T) **in**
  if $k(e) < k(u)$ **then** JOIN(INSERT$(L, e), e(u), R)$
  else if $k(u) < k(e)$ **then** JOIN$(L, e(u),$ INSERT$(R, e))$
  else JOIN(L, e, R)

UNION(T_1, T_2) =
 if $T_1 =$ LEAF **then** T_2
 else if $T_2 =$ LEAF **then** T_1
 else let $(L_2, u_2, R_2) =$ EXPOSE(T_2);
  $(L_1, b, R_1) =$ SPLIT$(T_1, k(u_2))$;
  $T_L =$ UNION(L_1, L_2) || $T_R =$ UNION(R_1, R_2);
  in JOIN$(T_L, e(u_2), T_R)$

INTERSECT(T_1, T_2, σ) =
 if $T_1 =$ LEAF **then** LEAF
 else if $T_2 =$ LEAF **then** LEAF
 else let $(L_2, u_2, R_2) =$ EXPOSE(T_2);
  $(L_1, b, R_1) =$ SPLIT$(T_1, k(u_2))$;
  $T_L =$ INTERSECT(L_1, L_2) || $T_R =$ INTERSECT(R_1, R_2);
  in if $b \neq \square$ **then** JOIN$(T_L, e(u_2), T_R)$
  else JOIN2(T_L, T_R)

FOR ORDERED MAPS:

UNION(T_1, T_2, σ) =
 if $T_1 =$ LEAF **then** T_2
 else if $T_2 =$ LEAF **then** T_1
 else let $(L_2, u_2, R_2) =$ EXPOSE(T_2);
  $(L_1, u_1, R_1) =$ SPLIT$(T_1, k(u_2))$;
  $T_L =$ UNION(L_1, L_2) || $T_R =$ UNION(R_1, R_2);
  in if $u_1 = \square$ **then** JOIN$(T_L, e(u_2), T_R)$
  else JOIN$(T_L, (k(u_1), \sigma(v(u_1), v(u_2))), T_R)$

Figure 4.1: Implementing UNION, INTERSECT, INSERT, SPLIT, and JOIN2 with just JOIN. The || notation indicates the recursive calls can run in parallel. The functions UNION and INTERSECT are slight variants of the algorithms described by Adams [1], although he did not consider parallelism, and only consider weight-balanced trees.

This chapter describes algorithms for various functions built atop JOIN for sequences, ordered sets and ordered maps. The pseudocodes for the algorithms in this section is shown in Figure 4.1. The formal definition along with the algorithms on augmented maps will be shown in Chapter 5.

Generally, beyond JOIN, the only access to the tree that the algorithms in this chapter make use of is through EXPOSE$(\cdot)$, $e(\cdot)$, SIZE$(\cdot)$ and NEWNODE$(\cdot)$. For algorithms on sequences, $k(\cdot)$ is also used. The ordered set in addtion uses $<_K$ and the ordered map also uses $v(\cdot)$. All these functions basically only read the root and we assume they all take constant time. Note that this may not be true for $<_K$ and NEWNODE for very complicated key types, but we assume so for the convinience of theoretical analysis.

4.1 Algorithms for Sequences

Typically, a sequence $s = \mathbb{SQ}(K)$ supports functions as listed in Table 4.1. Some of the notations are defined in Chapter 3. In the table n denotes the size of s (or $|s|$). Recall that in sequences, each key is an entry, and the entry type E is the key type K.

EMPTY	:	$\emptyset$, or $\{\}$, for trees it is a LEAF
DOMAIN(s)	:	$\{k(e) : e \in s\}$
SIZE(s)	:	$\|s\|$
SINGLETON(e)	:	$\{e\}$
SEQFROMLIST(l)	:	**Argument:** list $l = [e_1, e_2, \ldots, e_n], e_i \in E$ sequence s, where $s[i] = e_i$
TOLIST(s)	:	list $[s[1], s[2], \ldots, s[n]]$ ordered by indices
SELECT(s, i)	:	$s[i]$
FIRST(s)	:	$s[1]$ **if** $s \neq \emptyset$ **else** $\square$
LAST(s)	:	$s[n]$ **if** $m \neq \emptyset$ **else** $\square$
DELETEAT(s, i)	:	$\{s[1..i-1], s[i+1..n]\}$
INSERTAT(s, i, e)	:	$\{s[1..i-1], e, s[i..n]\}$
INDUPTO(s, i)	:	$\{s[1..i]\}$
INDDOWNTO(s, i)	:	$\{s[i..n]\}$
INDRANGE(s, i, j)	:	$\{s[i..j]\}$
JOIN2(s_1, s_2)	:	$\{s_1[1..\|s_1\|], s_2[1..\|s_2\|]\}$
SPLITAT(s, i)	:	$\langle\{$**INDUPTO**$(s, i-1), s[i],$ **INDDOWNTO**$(s, i+1)\}\rangle$
FILTER(s, ψ)	:	**Argument** $\psi : E \to$ **BOOL** $\{e \in s \mid \psi(e)\}$
MAPREDUCE $(s, f', g', b_\emptyset)$	:	**Argument** $g' : E \to B, f' : B \times B \mapsto B, b_\emptyset \in B$ is the identity of f $f(b_\emptyset, g'(s[1]), g'(s[2]), \ldots, g'(s[n]))$,

Table 4.1: The core functions on sequences. Throughout the table $i, j \in \mathbb{N}$, and $e, e_i \in E$, m, m_1, m_2 are ordered maps. s is a sequence. B is a type. $|\cdot|$ denotes the cardinality of a set. $\square$ represents an empty element.

Join2. JOIN2(T_L, T_R) returns a binary tree for which the in-order values is the concatenation of the in-order values of the binary trees T_L and T_R (the same as JOIN but without the middle key). For BSTs, all keys in T_L have to be less than keys in T_R. JOIN2 first finds the last entry e (by following the right spine) in T_L and on the way back to root, joins the subtrees along the path, which is similar to SPLIT

T_L by k. We denote the result of dropping k in T_L as T'_L. Then JOIN(T'_L, k, T_R) is the result of JOIN2. Unlike JOIN, the work of JOIN2 is proportional to the rank of both trees since both SPLIT and JOIN take at most logarithmic work.

Theorem 4.1.1. *The work of* JOIN2(T_L, T_R) *is* $O(r(T_L) + r(T_R))$ *for all balancing schemes described in Chapter 3 (bounds are w.h.p for treaps).*

Other Functions. All functions in Table 4.1 should be supported by a complete sequence interface. For example, the **MAPREDUCE**$(m, g', f', b_\emptyset)$ function applies the function g' to each element of the sequence m, and then sums them with the associative function f' with identity $b_\emptyset$. The **FILTER**(m, ψ) function picks up all entries in s satisfying indicator function ψ. All these functions can be implemented simply and generically with JOIN. Due to page limitation we omit most of them.

4.2 Algorithms for Sets

FROMLIST(l)	:	**Argument** $l = [e_1, \ldots, e_n], e_i \in E$ $\{e \mid e \in l\}$ ordered by $<_K$ on K
FIND(st, k)	:	$e \in st : k(e) = k$ **if** $k \in st$ **else** $\square$
DELETE(st, k)	:	$\{e \in st \mid k(e) \neq k\}$
INSERT(st, e)	:	**DELETE**$(st, k(e)) \cup \{e\}$
NEXT(st, k)	:	**FIRST**$(\{e \in st \mid k(e) > k\})$
PREVIOUS(st, k)	:	**LAST**$(\{e \in st \mid k(e) < k\})$
UPTO(st, k)	:	$\{e : e \in st \mid k(e) \leq k\}$
DOWNTO(st, k)	:	$\{e : e \in st \mid k(e) \geq k\}$
SPLIT(st, k)	:	$\langle \{e \in st \mid k(e) < k\}$, **FIND**$(m, k)$, $\{e \in m \mid k < k(e)\}\rangle$
RANGE(st, k_1, k_2)	:	$\{e \in st \mid k_1 \leq k(e) \leq k_2\}$
INTERSECTION(st_1, st_2)	:	$\{e \in st_1 \mid k(e) \in$ **DOMAIN**$(st_2)\}$
DIFFERENCE(st_1, st_2)	:	$\{e \in st_1 \mid k(e) \notin$ **DOMAIN**$(st_2)\}$
UNION(st_1, st_2, σ)	:	**DIFFERENCE**$(st_1, st_2) \cup$ **DIFFERENCE** (st_2, st_1) $\cup$ **INTERSECTION** (st_1, st_2, σ)

Table 4.2: The core functions on ordered sets. Throughout the table $k, k_1, k_2, k' \in K$ and $e, e_i \in E$, st, st_1, st_2 are ordered sets. s is a sequence. B is a type. $|\cdot|$ denotes the cardinality of a set. $\square$ represents an empty element.

All functions on sequences are valid on ordered sets. In addition, functions as listed in Table 4.2 are also in the ordered set $\mathbb{OS}(K, <_K)$ interface, and the comparison function $<_K$ should be valid on key type K. On ordered sets, the entry type E is equivalent as the key type K.

Split. For a BST T and key k, SPLIT(T, k) returns a triple (T_L, b, T_R), where T_L (T_R) is a tree containing all keys in T that are less (larger) than k, and b is a flag indicating whether $k \in T$. The algorithm first searches for k in T, splitting the tree along the path into three parts: keys to the left of the path, k itself (if it exists), and keys to the right. Then by applying JOIN, the algorithm merges all the subtrees on the left side (using keys on the path as intermediate nodes) from bottom to top to form T_L, and merges the right parts to form T_R. Figure 4.2 gives an example.

Theorem 4.2.1. *The work of* SPLIT(T, k) *is* $O(\log |T|)$ *for all balancing schemes described in Chapter*

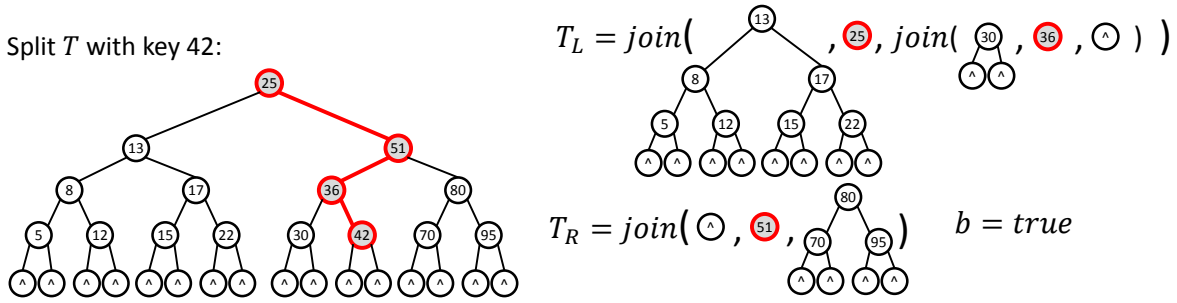

Figure 4.2: An example of SPLIT in a BST with key 42.

3 (w.h.p. for treaps). The two resulting trees T_L and T_R will have rank at most $r(T)$.

Union, Intersect and Difference. UNION(T_1, T_2) takes two BSTs and returns a BST that contains the union of all keys. The algorithm uses a classic divide-and-conquer strategy, which is parallel. At each level of recursion, T_1 is split by $k(T_2)$, breaking T_1 into three parts: one with all keys smaller than $k(T_2)$ (denoted as L_1), one in the middle either of only one key equal to $k(T_2)$ (when $k(T_2) \in T_1$) or empty (when $k(T_2) \notin T_1$), the third one with all keys larger than $k(T_2)$ (denoted as R_1). Then two recursive calls to UNION are made in parallel. One unions $L(T_2)$ with L_1, returning T_L, and the other one unions $R(T_2)$ with R_1, returning T_R. Finally the algorithm returns JOIN$(T_L, k(T_2), T_R)$, which is valid since $k(T_2)$ is greater than all keys in T_L are less than all keys in T_R.

The functions INTERSECT(T_1, T_2) and DIFFERENCE(T_1, T_2) are similar to UNION but in addition use JOIN2. The base cases are also different in the obvious way.

Theorem 4.2.2. *For all four balance schemes mentioned in Section 3, the work and span of the algorithm (as shown in Figure 4.1) of* UNION, INTERSECT *or* DIFFERENCE *on two balanced BSTs of sizes m and n ($n \geq m$) is $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ and $O(\log n \log m)$ respectively (the bound is in expectation for treaps).*

A generic proof of Theorem 4.2.2 working for all the four balancing schemes is shown in [21].

The work bound for these algorithms is optimal in the comparison-based model. In particular considering all possible interleavings, the minimum number of comparisons required to distinguish them is $\log\binom{m+n}{n} = \Theta\left(m \log\left(\frac{n}{m}+1\right)\right)$ [53].

Other Functions. All functions in Table 4.2 should be supported by a complete ordered set interface. For example, the **RANGE**(m, k_1, k_2) extracts the part of the map between keys k_1 and k_2, inclusive, returning a new set. The INSERT adds a new entry into the set. The pseudocode for INSERT is given in Figure 4.1. For a tree of size n it takes $O(\log n)$ work.

4.3 Algorithms for Ordered Maps

All functions on sets as listed in Table 4.2 are valid on ordered maps. Based on ordered sets, ordered maps introduce the value in the entry, so many of set functions (e.g., UNION, INTERSECT, INSERT and FROMLIST) can be assigned a function σ to combine values on maps. We call this function the complementary function. For an ordered map type $\mathbb{OM}(K, <_K, V)$, these functions are defined in Table 4.3. In the reasonable scenarios when two entries with the same key should appear simultaneously in the map, their values would be combined by the function σ. For example, when the same key appears

in both maps involved in a UNION or INTERSECT, the key will be kept as is, but their values will be combined by σ to be the new value in the result. Same case happens when an entry is inserted into a map that already has the same key in it, or when we build a map from a sequence that has multiple entries with the same key. A common scenario where this is useful for example, is to keep a count for each key, and have UNION and INTERSECT sum the counts, INSERT add the counts, and FROMSEQ keep the total counts of the same key in the sequence.

Union. The pseudocode for the UNION function with a complementary function is shown in Table 4.1. Its only difference from the set UNION function is that when SPLIT finds T_2's root in T_1, their values will be combined by σ (the last line of code).

$\textbf{FROMLIST}(l, \sigma)$	:	**Argument** list $l = \langle e_1, \ldots, e_n \rangle, e_i \in E$ $\{(k', v_{k'}) \mid \forall k' \in \textbf{DOMAIN}(l), v_{k'} = \sigma(v_{i_1}, v_{i_2}, \ldots v_{i_{n'}}) :$ $\forall v_{i_j}, (k', v_{i_j}) \in s, i_1 < i_2 < \cdots < i_{n'}\}$, ordered by $<_K$ on K
INSERT		**Argument** $\sigma : V \times V \to V$
(m, e, σ)	:	$\textbf{DELETE}(m, k(e)) \cup \{(k(e), \sigma\,(v(e), v(\textbf{FIND}(m, k(e)))))\}$
INTERSECTION		**Argument** $\sigma : V \times V \to V$
(m_1, m_2, σ)	:	$\{(k, \sigma(v(\textbf{FIND}(m_1, k)), v(\textbf{FIND}(m_2, k))))\mid k \in \textbf{DOMAIN}(m_1) \cap \textbf{DOMAIN}(m_2)\}$
UNION		**Argument** $\sigma : V \times V \to V$
(m_1, m_2, σ)	:	$\textbf{DIFFERENCE}(m_1, m_2) \cup \textbf{DIFFERENCE}\ (m_2, m_1) \cup \textbf{INTERSECTION}\ (m_1, m_2, \sigma)$

Table 4.3: The core functions on ordered maps. Throughout the table $k, k' \in K$, $v, v_i \in V$ and $e, e_i \in E$, m, m_1, m_2 are ordered maps. l is a list.

5 Augmented Maps: Definitions, Interfaces and Algorithms

Augmented Maps and Augmented Values. *Augmented maps*, as defined here, are structures that associate an ordered *map* with a "sum" (the *augmented value*) over all entries in the map. It is achieved by using the base function g, which gives the augmented value of a single element, and a combine function f which combines multiple augmented values, giving the augmented value of the map. More formally, an augmented map type $\mathbb{AM}(K, <_K, V, A, g, f, a_\emptyset)$ is parameterized on the following seven parameters:

K,	key type
$<_K \ : K \times K \to \textbf{BOOL}$,	total ordering on the keys
V,	value type
A,	augmented value type
$g : K \times V \to A$,	the base function
$f : A \times A \to A$,	the combine function
$a_\emptyset : A$	identity for f

The first three parameters correspond to a standard ordered map, and the last four are for the augmentation. f must be associative ($(A, f, a_\emptyset)$ is a monoid).

Then the formal definition of the *augmented value* of a map is given as follows:

Definition 5.0.1 (Augmented Value)**.** *Given a map $m = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$ over keys K and values V, a base function $g : K \times V \mapsto A$ and a combine $f : A \times A \mapsto A$, its* augmented value *is*

$$\mathcal{A}(m) = f(g(k_1, v_1), g(k_2, v_2), \ldots, g(k_n, v_n)) \tag{5.1}$$

Implement Augmented Maps with Augmented Trees. Augmented maps are independent of representation. One possible and efficient implementation is to use augmented trees [33, 44] storing partial sums in subtrees. An augmented tree structure is a search tree (possibly but not necessarily binary) in which each node is augmented with a value recording some information about its subtree. Although this concept is widely-used in textbooks [33], the definition is often used in a more general way to mean any form of data associated with the nodes of a tree. This thesis uses augmented trees to implement augmented maps, and gives a definition specific to implementing maps. Each entry (a *key* and a *value*) is stored in a tree node, and tree nodes are sorted by keys. Each node also keeps track of the *augmented value* of the sub-map rooted at it. These partial sums are useful to make many functions more efficient. The formal definition of *augmented tree* for augmented maps is given as follows:

Definition 5.0.2 (Augmented Tree)**.** *An augmented tree T for a map M is a search-tree associated with two functions f and g, for which each node u is augmented with an value $\mathcal{A}(M_u)$ (see in Definition 5.0.1), where $M_v \subseteq M$ is the range of M belonging to the subtree rooted at v.*

A (sub)tree in an augmented tree consists of its left subtree, the root, and its right subtree. The augmented value of this tree, by definition, should combine the augmented value of the three components in order, i.e. the augmented values can be maintained by $\mathcal{A}(u) = f(\mathcal{A}(C_L(u)), g(k(u), v(u)), \mathcal{A}(C_R(u)))$.

The Interface for Augmented Maps. An augmented map type supports an interface with standard functions on ordered maps as well as a collection of functions that make use of f and g. All functions in Table 4.1, 4.2 and 4.3 should be supported by augmented map interface.

Function	**Definition**	**Condition**
$\textbf{AUGVAL}(m)$	$\mathcal{A}(m) = \textbf{MAPREDUCE}(m, g, f, a_\emptyset)$	
$\textbf{AUGLEFT}(m, k)$	$\textbf{AUGVAL}(\textbf{UPTO}(m, k))$	
$\textbf{AUGRIGHT}(m, k)$	$\textbf{AUGVAL}(\textbf{DOWNTO}(m, k))$	
$\textbf{AUGRANGE}(m, k_1, k_2)$	$\textbf{AUGVAL}(\textbf{RANGE}(m, k_1, k_2))$	
$\textbf{AUGFILTER}(m, \psi)$	**Argument** $\psi : A \mapsto \textbf{BOOL}$	$\psi' : E \mapsto \textbf{BOOL}, \psi'(e) \Leftrightarrow \psi(g(e))$
	$\textbf{FILTER}(m, \psi')$	$\forall a, b \in A, \psi(a) \vee \psi(b) \Leftrightarrow \psi(f(a, b))$
$\textbf{AUGPROJECT}$	**Argument** $g' : A \mapsto B, f' : B \times B \mapsto B$	$(B, f', g'(a_\emptyset))$ is a monoid.
(m, g', f', m, k_1, k_2)	$g'(\textbf{AUGRANGE}(m, k_1, k_2))$	$f'(g'(a), g'(b)) = g'(f(a, b))$

Table 5.1: Core functions on augmented ordered maps. In the table we assume $k, k_1, k_2 \in K$, and $e \in K \times V$, m is an augmented map.

Most important are the definitions in Table 5.1, which are functions specific to augmented maps. All of them can be computed using the plain ordered map functions. However, they can be much more efficient by maintaining the augmented values of sub-maps (partial sums) in a tree structure. They also benefit from using the augmenting functions f and g as operands for plain ordered map functions. The function $\textbf{AUGVAL}(m)$ returns $\mathcal{A}(m)$, which is equivalent to $\textbf{MAPREDUCE}(m, g, f, a_\emptyset)$ but can run in constant instead of linear work. This is because the augmented value of the whole map is stored in the tree root, and can be maintained during updates. The function $\textbf{AUGRANGE}(m, k_1, k_2)$ is equivalent to $\textbf{AUGVAL}(\textbf{RANGE}(m, k_1, k_2))$ and $\textbf{AUGLEFT}(m, k)$ is equivalent to $\textbf{AUGVAL}(\textbf{UPTO}(m, k))$. These can also be implemented efficiently using the partial sums.

The last two functions in Table 5.1 accelerate three common queries on augmented maps, but are only applicable when their function arguments meet certain requirements. They also can be computed using the plain map functions, but can be much more efficient when applicable because their input operands satisfy some conditions related to the augmenting functions g and f. The $\textbf{AUGFILTER}(m, \psi)$ function is equivalent to $\textbf{FILTER}(m, \psi')$, but is only applicable if $\psi'(e) \Leftrightarrow \psi(g(e))$ and $\psi(a) \vee \psi(b) \Leftrightarrow \psi(f(a, b))$ for any $e \in E$ and $a, b \in A$ ($\vee$ is the logical or). In this case the **FILTER** function can make use of the partial sums. For example, when f is a logical-or and $\psi(a)$ simply returns a, we can filter out a whole subtree once we see it has `false` as its augmented value instead of traversing all entries one by one. The function is used in interval trees (Section 7.1) as well as the application of inverted indices (Section 7.3). The $\textbf{AUGPROJECT}(m, g', f', k_1, k_2)$ function is equivalent to $g'(\textbf{AUGRANGE}(m, k_1, k_2))$. It requires, however, that $(B, f', g'(a_\emptyset))$ is a monoid and that $f'(g'(a), g'(b)) = g'(f(a, b))$. This function projects the augmented value of a sub-map on m over a range to type B. This function is useful when computing $AugRange(m, k_1, k_2)$ is expensive, e.g., the augmented values are themselves complicated structures such as maps or other large data structures. It allows projecting the partial augmented values down onto another type by g' (e.g. project augmented values that are augmented maps to simple values like their sizes) then summing them by f', and is much more efficient when applicable. For example in range trees where each augmented value is itself an augmented map, it is used to answer queries by projecting the inner trees to their weight sums, and greatly improves the performance.

Algorithms for Augmented Maps. The augmentation is implemented by storing with every tree node the augmented sum of the subtree rooted at that node. This localizes application of the augmentation functions f and g to when a node is created or updated with new children (e.g., in rotations)[1]. In our algorithms, all creation and updating of nodes are handled in JOIN, which also deals with rebalancing when needed. Therefore all the algorithms and code except JOIN which do not explicitly need the augmented value are unaffected by and even oblivious to augmentation. All algorithms on sequences, ordered sets and ordered maps as shown in Table 4.1, 4.2 and 4.3 can be directly applied to augmented maps. The corresponding bounds still hold assuming that f, g and $a_\emptyset$ return in constant time. Two algorithms (AUGFILTER and AUGLEFT) are given as examples of augmented map functions.

Filter and AugFilter. The Filter and augFilter function both select all entries in the map satisfying condition ψ'. For a (non-empty) tree T, FILTER recursively filters its left and right branches in parallel, and combines the two results with JOIN or JOIN2 depending on whether ψ' is satisfied for the entry at the root. It takes linear work and $O(\log^2 n)$ depth for a balanced tree. The AUGFILTER function has the same effect as FILTER, but is only applicable if there exists a function $\psi : A \mapsto \textbf{BOOL}$ that $\psi'(e) \Leftrightarrow \psi(g(e))$ and $\psi(a) \vee \psi(b) \Leftrightarrow \psi(f(a,b))$. This can asymptotically improve efficiency since if $\psi(\mathcal{A}(T))$ is false, then we know that ψ will not hold for any entries in the tree T, so the search can be pruned (see Figure 5.1). It takes $O(k \log(n/k+1))$ work and $O(\log^2 n)$ depth, and is significantly more efficient when k is small.

Reporting Augmented Values. As an example, we give the algorithm of AUGLEFT(T, k') in Figure 5.1, which returns the augmented value of all entries with keys less than k'. It compares the root of T with k', and if k' is smaller, it calls AUGLEFT on its left subtree. Otherwise the whole left subtree and the root should be counted. Thus we directly extract the augmented value of the left subtree, convert the entry in the root to an augmented value by g, and recursively call AUGLEFT on its right subtree. The three results are combined using f as the final answer. This function visits at most $O(\log n)$ nodes, so it costs $O(\log n)$ work and depth assuming f, g and $a_\emptyset$ return in constant time. The AUGRANGE function, which reports the augmented value of all entries in a range, can be implemented similarly with the same asymptotical bound.

AUGFILTER$(T, \psi) =$
 if $(T = \emptyset)$ OR NOT$(\psi(\mathcal{A}(T))$ **then** $\emptyset$
 else let $(L, u, R) = T$
 and $L' =$ AUGFILTER(L, ψ) ||
 $R' =$ AUGFILTER(R, ψ)
 in if $\psi(g(e(u)))$ **then** JOIN(L', u, R')
 else JOIN2(L', R')

AUGLEFT$(T, k') =$
 if $T = \emptyset$ **then** $a_\emptyset$
 else let $(L, u, R) = T$
 in if $k' < k(u)$ **then** AUGLEFT(L, k')
 else $f(\mathcal{A}(L), g(e(u))$,AUGLEFT$(R, k'))$

Figure 5.1: Algorithms on augmented maps.

[1]For supporting persistence, updating a tree node, e.g., when it is involved in rotations and is set a new child, usually results in the creation of a new node. See details in the persistence part.

6 The PAM Library

We have implemented augmented maps as part of the PAM library. The library has several interesting features beyond augmentation which we overview here. All these features can be combined with augmentation, and indeed are mostly orthogonal.

PAM uses a reference counting garbage collection maintaining nodes at the trees. This technique stores the number of references or pointers to each tree node. This is useful in maintaining many properties mentioned in this section.

In PAM, we use AVL trees [2], but as mentioned, the same algorithms can also be applied to at least three other balancing schemes [21]: red-black (RB) trees [12], weight-balanced (WB) trees and treaps [86], with very small changes in the PAM code.

Persistence The PAM library fully supports persistence (or functional) [41]. With persistence any update to a data structure creates a new version, while maintaining old versions in main memory—i.e. the old versions survive persistently in main memory until there are no pointers to it anymore, at which point it can be garbage collected. In a persistent implementation of maps, for example, an insertion will create a new map, with the element added, while keeping the old version. Such a history persistent insertion need not copy the whole map, but instead can be implemented in $O(\log n)$ time by creating a new path to the position of the insert, without making any changes to the old path. Such *path copying* is the default implementation in functional languages, where data cannot be overwritten [74]. This requires no versioning or other auxiliary fields internally on the data structure. Path copying, however, requires that parts of the trees are shared by multiple maps, and therefore requires a garbage collector to avoid deleting shared components or keeping unreachable components.

Persistence has at least two important advantages. Firstly it has many applications in developing efficient data structures [41], and this was also its original motivation. Many examples of the use of augmented maps in this thesis (such as maintaining inverted indices and range trees) also use persistence in an critical way. Secondly, persistence can be useful for supporting the snapshot isolation in database, which will be one of the future directions of this thesis (see Section 9.1).

In PAM we implement persistence on maps using path copying and a reference counting collector. We additionally implement the optimization that if updating the only reference, i.e., the reference count is one, then a node is reused rather than being copied. This optimization makes almost a factor of two improvement in runtimes, and in our experiments can be applied in most common cases. In our experiments we show that insert with overwrites is as fast as the STL version of non-persistent inserts. Importantly from the perspective of this paper is that such copying is fully consistent with the maintenance of augmented data. In particular when a path is copied, augmented values are updated along the path in the same way, and at the same cost as, they would be updated if not copied in traditional imperative implementations of trees.

Safe for Concurrency PAM is designed to be safe for concurrency such that any number of threads can safely update their copy of a map at the same time even though their map data can be shared. All operations are lock-free when enough memory is pre-allocated, otherwise it is at the mercy of the OS. Threads interact through shared tree nodes and the memory manager. The main interaction on nodes is

through updating and checking reference counts. This is done with a fetch-and-add, which in turn uses a compare-and-swap, which is lock-free. The memory manager maintains separate pools of nodes for each thread, along with a shared lock-free stack-based pool. Blocks of 64K tree nodes are taken and put into the shared pool. If the shared pool runs out of blocks then a larger segment is allocated from the OS.

Significant care was needed to ensure operations are done in the right order to be safe for concurrency. For example, when copying a node it is important that the reference counts on the children are incremented first before the reference count on the old parent is decremented. In a sequential version this is not important.

7 Applications

7.1 Interval Trees

We give an example of our interface on the interval trees [33, 34, 34, 42, 44, 59, 69]. This data structure maintains a set of intervals on the real line, each defined by a left and a right endpoint. Various queries can be answered, such as stabbing queries that determines whether a given point is in an interval, or returning all intervals that contain some point.

In an interval tree, each interval is stored in a tree node, sorted by the left endpoint (key). We set the value to be the right endpoint. When stabbing a point p, we look at the maximum value among all tree nodes with keys less than p, and if it is more than p, then p is covered by some interval in the tree. To answer this efficiently, each tree node is associated with a value that is the maximum right endpoint among all intervals in its subtree. This associated value can be easily maintained using the augmented value under the augmented map framework using a combine function $\max$. An example is shown in Figure 7.1. The definition using augmented map abstraction is:

$$I = \mathbb{AM}(\mathbb{R}, <_{\mathbb{R}}, \mathbb{R}, \mathbb{R}, (k, v) \mapsto v, \max_{\mathbb{R}}, -\infty)$$

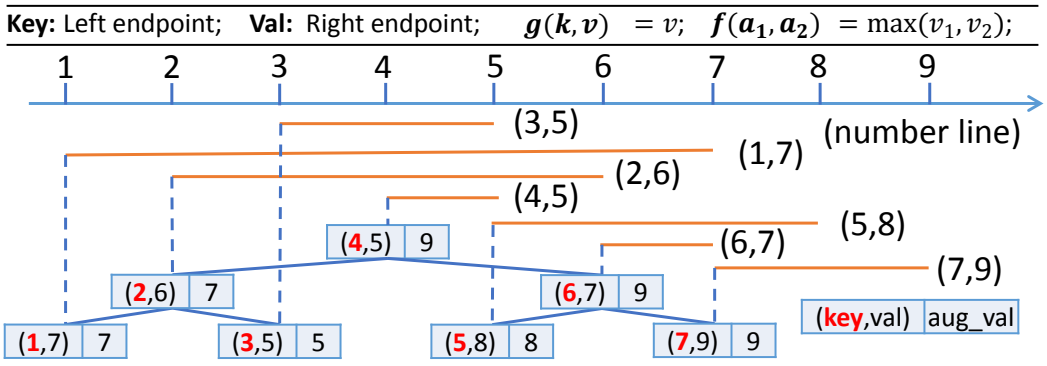

Figure 7.1: An example of an interval tree.

It is also easy to implement it by code using PAM. Figure 7.2 shows the C++ code of the interval tree structure using PAM. The augmentation is defined in `aug` from line 4, containing the augmented value type (`aug_t`), the base function (`g`), the combine function (`f`), and the identity of f (`identity`). An augmented map (line 10) is then declared as the interval tree structure with specified types of key (`P`), value (`P`), augmentation (`aug`) and the comparison function (`less`). The constructor on line 12 builds an interval tree from an array of n intervals by directly calling the augmented-map constructor in PAM ($O(n \log n)$ work). The function `stab(p)` returns if p is inside any interval using `m.aug_left(`k`)`. As defined in Chapter 5, this function returns the augmented sum, which is the $\max$ on values, of all entries with keys less than k. As mentioned we need only to compare it with p as done in `stab`.

```
1 struct interval_map {
2   using P = int;
3   using interval = pair<P, P>;
4   struct aug {
5     using aug_t = P;
6     static aug_t identity() {return -inf;}
7     static aug_t base(P k, P v) {return v;}
8     static aug_t combine(aug_t a, aug_t b) { return max(a,b);}  };
9   auto less= [] (P a, P b) {return a<b;}
10  using i_tree = aug_map<P,P,aug,less>;
11  i_tree m;
12  interval_map(interval* A, int n) { m.fromSeq(A,A+n); }
13  bool stab(P p) {return (m.aug_left(p)>p);} };
```

Figure 7.2: The definition of interval maps using PAM in C++.

7.2 Range Trees

Given a set of n points $P = \{p_i = (x_i, y_i)\}$ in the plane, each with weight $w_i \in W$, a *2D range query* asks for the sum of weights of points within a rectangle defined by a horizontal range (x_L, x_R) and vertical range (y_L, y_R). In this section, we describe how to adapt 2D range tree to the PAM framework to address this problem.

The standard 2D range tree [15, 69, 85] is a two-level tree (or map) structure. The outer level stores all the points ordered by the x-coordinate. Each tree node stores an inner tree with all points in its subtree but ordered by the y-coordinate. Then a range query can be done by two nested queries on x- and y-coordinates respectively. Sequential construction time and query time is $O(n \log n)$ and $O(\log^2 n)$ respectively (the query time can be reduced to $O(\log n)$ with reasonably complicated approaches).

The outer tree (R_O) is an augmented map in which keys are points (sorted by x-coordinate) and values are weights. The augmented value, which is the inner tree, is another augmented tree (R_I) storing the coordinates as the key (sorted by y-coordinate) and the weight as the value. Then the augmented value of the inner tree accumulates the weights in the subtree. The inner tree of each node stores all the nodes in its two children and itself, thus the base and combine function of the outer tree can be defined as a singleton of R_I and a UNION on two inner trees. The definition of a range tree under the augmented map framework is:

$R_I = \mathbb{AM}(P, <_Y, W, W, (k, v) \to v, \quad +_W, 0_W)$

$R_O = \mathbb{AM}(P, <_X, W, R_I, R_I.\text{singleton}, \cup, \quad \emptyset \quad)$

Here P is the point type. W is the weight type. $+_W$ and 0_W are the addition function on W and its identity respectively.

It is worth noting that because of the persistence of PAM library, the combine function UNION does not affect the inner trees in its two children, but builds a new version of R_I containing all the elements in its subtree. This is very important in guaranteeing the correctness of the algorithm.

To answer the query, we conduct two nested range searches (x_L, x_R) on the outer tree and (y_L, y_R) on the related inner trees [69, 85]. It can be implemented using the augmented map functions as:

QUERY$(r_O, x_L, x_R, y_L, y_R) =$
  let $g'(r_I) =$ AUGRANGE(r_I, y_L, y_R)
  in AUGPROJECT$(g', +_W, r_O, x_L, x_R)$

The **AUGPROJECT** on R_O is the top-level searching of x-coordinates in the outer tree, and g' projects the inner trees to the weight sum of the corresponding y-range. f' $(+_W)$ combines the weight of all results of g' to give the sum of weights in the rectangle. When f' is an addition, g' returns the range sum, and f is a UNION, the condition $f'(g'(a), g'(b)) = g'(a) + g'(b) = g'(a \cup b) = g'(f(a, b))$ holds, so AUGPROJECT is applicable. Combining the two steps together, the query time is $O(\log^2 n)$. We can also answer range queries that report all point inside a rectangle in time $O(k + \log^2 n)$, where k is the output size.

7.3 Ranked Queries on Inverted Indices

The third application of augmented maps is for building and searching a weighted inverted index of the kind used by search engines [81, 96] (also called an inverted file or posting file). For a given corpus, the index stores a mapping from words to second-level mappings. Each second-level mapping, maps each document that the term appears in to a weight, corresponding to the importance of the word in the document and the importance of the document itself. Using such a representation, conjunctions and disjunctions on terms in the index can be found by taking the intersection and union, respectively, of the corresponding maps. Weights are combined when taking unions and intersections. It is often useful to only report the top k results, as a search engine would list on its first page of results.

This can be represented rather directly in our interface. The inner map, maps document-ids (D) to weights (W) and uses maximum as the agumenting function f. The outer map maps terms (T) to inner maps. This corresponds to the maps:

$M_I = \mathbb{AM}(D, W, <_D, W, (k, v) \to v, \max_W, 0)$

$M_O = \mathbb{OM}(T, M_I, <_T)$

We use UNION and INTERSECT on ordered maps that allows passing a complementary function (see Chapter 6) for combining weights. The AUGFILTER function can be used to select the k best results after taking unions and intersections over terms. Note that an important feature is that the UNION function can take time that is much less that the size of the output (e.g., see Section 4). Therefore using augmentation can significantly reduce the cost of finding the top k relative to naively checking all the output to pick out the k best. The C++ code for our implementation is under 50 lines.

8 Preliminary Experimental Results

Some experiments have already been conducted to test the performance of the PAM library. For the experiments we use a 72-core Dell R930 with 4 x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache), and 1Tbyte memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. Our code was compiled using g++ 5.4.1, which supports the Cilk Plus extensions. Of these we only use `cilk_spawn` and `cilk_sync` for fork-join, and `cilk_for` as a parallel loop. We compile with `-O2`.

Here we show experimental results that measure performance of four applications: the augmented sum (augmented with the range sums based on addition), the interval tree, the 2D range tree and the word index searching. Table 8.1 gives the theoretical cost and running time for the four applications. The construction cost for all the four applications are $O(n \log n)$, which is optimal considering that each at least involves a sorting. For the simplest range sum application, our implementation can scale up to 10^{10} elements in the tree, and has a 80-fold speedup for construction and a 130-fold speedup for queries. The other three applications also achieve speedup of 40-50 for constructions, and 80+ for queries. More results can be found in our previous work [19, 21, 88].

Application	**In Theory (Asymptotical bound)**			**In Practice (Running Time in seconds)**							
	Construct		**Query**	**Construct**				**Query**			
	Work	**Span**		**Size**	**Seq.**	**Par.**	**Spd.**	**Size**	**Seq.**	**Par.**	**Spd.**
Range Sum	$O(n \log n)$	$O(\log n)$	$O(\log n)$	10^{10}	2155.24	27.10	79.53	10^8	429.08	3.22	133.13
Interval Tree	$O(n \log n)$	$O(\log n)$	$O(\log n)$	10^8	15.45	0.26	59.02	10^8	55.13	0.60	91.91
2d Range Tree	$O(n \log n)$	$O(\log^3 n)$	$O(\log^2 n)$	10^8	258.59	4.71	54.90	10^6	50.16	0.63	80.00
Inverted Index	$O(n \log n)$	$O(\log^2 n)$	*	1.96×10^9	540.00	13.7	39.40	10^5	401.00	5.62	74.70

Table 8.1: The asymptotic cost and experimental results of the applications using PAM. **Seq.** = sequential, **Par.** = Parallel (on 72 cores with 144 hyperthreads), **Spd.** = Speedup. "Work" and "Span" are used to evaluate the theoretical bound of parallel algorithms (see Section 3). *: Depends on the query.

9 Directions for Future Work

9.1 Database Snapshot Isolation

In PAM, maps are implemented in a persistent (or functional) manner, which means that any update will create a new version of the data structure, without destroying the old one. This property can be useful for efficient implementations of multiversion concurrency control (MVCC), which further can be used to support snapshot isolation [16] in database. This especially uses PAM's reference counting and GC for guaranteeing the correctness and efficiency.

Besides the snapshot isolation part, most of the regular database operations in transactions (e.g., `insert`, `read`, `update`, etc.) can be fulfilled using PAM's interface in a "batched" manner. For example, all insertions in a short period can be done by a UNION and commit to the database together.

(Potential topic 1). ***Use the interface of PAM library to implement an efficient database system supporting snapshot isolation and many other database operations. Apply it to real data, adopt special optimizations and test performance.***

9.2 Write-Efficient BST

The join-based implementation can take the advantage on batched operations and reduce redundant computations when dealing with a collection of keys, and is especially useful in the write-efficient setting. For example, when merging all elements in a smaller tree T_i into a large tree T (assume $|T| \gg |T_i|$, and $|T|$ is very large and probably exceed the cache size), we directly call UNION(T, T_i) to merge the smaller tree into the larger one. As shown in Figure 4.1, T_i will first be split by T's root, which also concentrates all read and write transfers on elements on T_i. These writes and reads are very cheap since the elements in T_i are frequently operated and are very likely to be in the cache. As to the JOIN function after the recursive calls, the two operands T_l and T_r are T's left and right subtree appended with some elements in T_i. Considering that $|T| \gg |T_i|$, the height (or size) of T_l and T_r should not differ much from T's original two subtrees. Thus JOIN only need a constant number of rotations (in other words, constant reads or writes) to rebalance the final result. As for the read transfers, besides all elements in T_i are frequently accessed and is very likely in the cache, the pattern to access (read) the elements in T is obvious—basically it is the same order of visited nodes in DFS, which also gives good locality.

(Potential topic 2). ***Test the performance (read/write rate) of the join-based functions. Which balancing scheme have the best performance assuming the ARAM model? Is there any optimizations that can make these algorithms more write-efficient?***

9.3 More Complete Interface

There are still many functions waiting to be added to the library. For example, the "lazily" insertion and deletion which postpone calling augmenting functions when inserting or deleting entries. In the current version, the INSERT function will call JOIN on each node u on the corresponding path. The JOIN function will cause new augmented values, which in turn results in invocations on the augmenting

functions. For some applications, especially those multi-level trees like range trees, the invocation on the combine functions (UNION) is expensive. However, if the combine function is communicative, the newly-added entry can be directly combined into the current augmented value of u (as long as no rebalancing is required). This can save much work for the INSERT function on complicated structures. When rebalancing is necessary, the combine function should be called anyway to generate new augmented values. Also the more upper level it happens, usually the more expensive it is. In this case the work cannot be saved. However one insertion is less likely to cause imbalance, and even less likely for the top levels. There are good grounds for believing the amortized cost of such a lazy insertion (when applicable) is reasonably low, but this needs further justification.

We also plan to add functions such as INTERSECT on two maps with the same key set but different value sets, which generally can be used to implement the "join" operation in databases.

(Potential topic 3). ***Add more functions to the library. This includes both the implementation and the rigorous proof for corresponding bounds.***

Also, the binary tree can also be used to support sequences, which also requires interfaces such as JOIN, SPLIT and JOIN2. Convenient interfaces for sequences can be used in many algorithms and some database systems. In the future I plan to add functions for supporting sequences in the library.

(Potential topic 4). ***Add an interface for supporting sequences using join-based algorithms.***

9.4 More Applications

Beyond the applications mentioned in this proposal, there are also many data structures and problems that can be adjusted to the augmented map framework. They include (write-efficient) priority trees, (write-efficient) interval trees[1], Merkle trees, and so on.

(Potential topic 5). ***Apply the augmented map framework and the PAM library to more applications.***

Also, unlike many static versions that only store data in the leaves, our range trees and segment trees allow rotations and rebalance, which make the tree dynamic. Especially with the lazy insertions and deletions introduced in section 9.3, the dynamic versions can be more efficient. In the future, adding functions for supporting dynamic versions for such trees is also important to make the applications more complete.

(Potential topic 6). ***Add functions for supporting dynamic versions of applications using PAM.***

[1]This is another version of interval trees different from what is introduced in 7.1.

Bibliography

[1] Stephen Adams. Implementing sets effciently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992. 4.1

[2] Georgy Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *Proc. of the USSR Academy of Sciences*, 145:263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962. 1, 3, 6

[3] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proc. ACM SIGMOD-SIGACT-SIGAI Symp. on Principles of Database Systems (PODS)*, pages 429–440, 2016. 2

[4] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988. 3

[5] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajech K. Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. In *Proc. USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011. 3

[6] Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. *arXiv preprint arXiv:1510.05433*, 2015. 2

[7] Ahmed M. Aly, Hazem Elmeleegy, Yan Qi, and Walid Aref. Kangaroo: Workload-aware processing of range data and range queries in hadoop. In *Proc. ACM International Conference on Web Search and Data Mining (WSDM)*, pages 397–406, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3716-8. 2

[8] Flurry analytics. URL `https://developer.yahoo.com/flurry/docs/analytics/`. 2

[9] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. Path processing using solid state storage. In *Proc. International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2012. 3

[10] Antonio Barbuzzi, Pietro Michiardi, Ernst Biersack, and Gennaro Boggia. Parallel bulk insertion for large-scale analytics applications. In *Proc. International Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010. 2

[11] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–369, 2017. 2

[12] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 1, 3, 6

[13] Naama Ben-David, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Parallel algorithms for asymmetric read-write costs. In *Proc. ACM*

Symposium on Parallelism in Algorithms and Architectures (SPAA), 2016. 3

[14] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.*, 29(7):571–577, July 1980. 1

[15] Jon Louis Bentley. Decomposable searching problems. *Information processing letters*, 8(5): 244–251, 1979. 1, 7.2

[16] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995. 3, 9.1

[17] Juan Besa and Yadran Eterovic. A concurrent red-black tree. *J. Parallel Distrib. Comput.*, 73(4): 434–449, 2013. 2

[18] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 16–26, 1998. 2

[19] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Parallel range and segment queries with augmented maps. 4, 5, 8, 8

[20] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Sorting with asymmetric read and write costs. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015. 3

[21] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–264, 2016. 1, 1, 3, 8, 4, 4.0.1, 4.2, 6, 8

[22] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *24th Annual European Symposium on Algorithms*, pages 14:1–14:18, 2016. 3

[23] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 443–454, 2016. 3

[24] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320, 1980. 3

[25] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1):202–229, 1998. 3

[26] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974. 3

[27] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 257–268, 2010. 2

[28] Trevor Brown and Hillel Avni. *Range Queries in Non-blocking k-ary Search Trees*, pages 31–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35476-2. 2

[29] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In

Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP), pages 329–342, 2014. 2

[30] Erin Carson, James Demmel, Laura Grigori, Nicholas Knight, Penporn Koanantakool, Oded Schwartz, and Harsha Vardhan Simhadri. Write-avoiding algorithms. In *IEEE International Parallel and Distributed Processing Symposium*, pages 648–658, 2016. 3

[31] Chee Yong Chan and Yannis E. Ioannidis. Hierarchical prefix cubes for range-sum queries. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 675–686, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1-55860-615-7. 2

[32] Sangyeun Cho and Hyunjin Lee. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009. 3

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (second edition)*. MIT Press and McGraw-Hill, 2001. 5, 7.1

[34] Costin S. C# based interval tree. `https://code.google.com/archive/p/intervaltree`, 2012. 7.1

[35] Bin Cui, Beng Chin Ooi, Jianwen Su, and Kian-Lee Tan. Contorting high dimensional data for efficient main memory knn processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM, 2003. 1

[36] Bin Cui, Beng Chin Ooi, Jianwen Su, and K-L Tan. Main memory indexing: the case for bd-tree. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):870–874, 2004. 1

[37] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959. 1

[38] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Hai H. Li, and Yiran Chen. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *Proc. ACM Design Automation Conference (DAC)*, 2008. 3

[39] Xiangyu Dong, Norman P. Jouupi, and Yuan Xie. PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM. In *Proc. ACM International Conference on Computer-Aided Design (ICCAD)*, 2009. 3

[40] Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356, 2014. 2

[41] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proc. ACM symposium on Theory of computing (STOC)*, pages 109–121, 1986. 3, 6

[42] Herbert Edelsbrunner. Dynamic rectangle intersrction searching. Technical Report Institute for Technical Processing Report 47, Technical University of Graz, Austria, 1980. 7.1

[43] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. In *Experimental Algorithms*, pages 111–122. Springer, 2014. 2

[44] Preparata Franco and Michael Ian Preparata Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 1985. 5, 7.1

[45] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960. 1

[46] Leonor Frias and Johannes Singler. Parallelization of bulk operations for STL dictionaries. In *Euro-Par 2007 Workshops: Parallel Processing, HPPC 2007, UNICORE Summit 2007, and VHPC 2007*, pages 49–58, 2007. 2

[47] Hong Gao and Jian-Zhong Li. Parallel data cube storage structure for range sum queries and dynamic updates. *J. Comput. Sci. Technol.*, 20(3):345–356, May 2005. ISSN 1000-9000. 2

[48] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, Mar 1997. 2

[49] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 47–57, New York, NY, USA, 1984. ACM. ISBN 0-89791-128-8. 2

[50] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *Proc. ACM International Conference on Management of Data (SIGMOD)*, pages 73–88, New York, NY, USA, 1997. ACM. ISBN 0-89791-911-4. 2

[51] HP. HP, SanDisk partner on memristor, ReRAM technology. http://www.bit-tech.net/news/hardware/2015/10/09/hp-sandisk-reram-memristor, October 2015. 3

[52] Jingtong Hu, Qingfeng Zhuge, Chun Jason Xue, Wei-Che Tseng, Shouzhen Gu, and Edwin Sha. Scheduling to optimize cache utilization for non-volatile main memories. *IEEE Transactions on Computers*, 63(8), 2014. 3

[53] Frank K. Hwang and Shen Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM J. on Computing*, 1(1):31–39, 1972. 4.2

[54] IBM. www.slideshare.net/IBMZRL/theseus-pss-nvmw2014, 2014. 3

[55] Intel. Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, July 2015. 3

[56] J Katajainen. Efficient parallel algorithms for manipulating sorted sets. In *Proceedings of the 17th Annual Computer Science Conference*. University of Canterbury, 1994. 2

[57] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D Nguyen, Tim Kaldewey, Victor W Lee, Scott A Brandt, and Pradeep Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010. 1

[58] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2014. 3

[59] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing intervals efficiently in object-relational databases. In *Proc. International Conference on Very Large Data Bases (VLDB)*, pages 407–418, 2000. 2, 7.1

[60] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980. 2

[61] Kim S. Larsen. AVL trees with relaxed balance. *J. Comput. Syst. Sci.*, 61(3):508–522, 2000. 2

[62] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proc. ACM International Symposium on Computer Architecture (ISCA)*, 2009. 3

[63] Tobin J Lehman and Michael J Carey. A study of index structures for main memory database management systems. In *Proc. VLDB*, 1986. 1

[64] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013. 1

[65] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, 2013. ISBN 978-1-4673-4909-3. 2

[66] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 302–313. IEEE, 2013. 1

[67] LevelDB. URL `leveldb.org`. 2

[68] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting nvm in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, page 2. ACM, 2015. 3

[69] Mark De Berg Mark, Mark Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry*. Springer, 2000. 2, 7.1, 7.2

[70] Jagan S. Meena, Simon M. Sze, Umesh Chand, and Tseung-Yuan Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9, 2014. 3

[71] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, 2014. 2

[72] Gabriele Neyer. dD range and segment trees. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017. URL `http://doc.cgal.org/4.10/Manual/packages.html`. 2

[73] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973. 1, 3

[74] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. 6

[75] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. 1

[76] Oracle. Oracle nosql. URL `https://www.oracle.com/database/nosql/index.html`. 2

[77] Heejin Park and Kunsoo Park. Parallel algorithms for red–black trees. *Theoretical Computer Science*, 262(1):415–435, 2001. 2

[78] Wolfgang J. Paul, Uzi Vishkin, and Hubert Wagener. Parallel dictionaries in 2-3 trees. In *Proc. Intl. Colloq. on Automata, Languages and Programming (ICALP)*, pages 597–609, 1983. 2

[79] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, 2012. 2

[80] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool, 2011. 3

[81] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets:*. Cambridge University Press, 10 2011. 7.3

[82] Jun Rao and Kenneth A Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, volume 99, pages 78–89, 1999. 1

[83] Jun Rao and Kenneth A Ross. Making b+-trees cache conscious in main memory. In *ACM SIGMOD Record*, volume 29, pages 475–486. ACM, 2000. 1

[84] RocksDB. URL `rockdb.org`. 2

[85] Hanan Samet. *The design and analysis of spatial data structures*, volume 199. Addison-Wesley Reading, MA, 1990. 7.2

[86] Raimund Seidel and Celcilia R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996. 1, 3, 6

[87] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985. 1

[88] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. *CoRR*, abs/1612.05665, 2016. URL `http://arxiv.org/abs/1612.05665`. 2, 3, 5, 8, 8

[89] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983. ISBN 0-89871-187-8. 1

[90] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973. 1

[91] Cong Xu, Xiangyu Dong, Norman P. Jouppi, and Yuan Xie. Design implications of memristor-based RRAM cross-point structures. In *Proc. IEEE Design, Automation and Test in Europe (DATE)*, 2011. 3

[92] Byung-Do Yang, Jae-Eun Lee, Jang-Su Kim, Junghyun Cho, Seung-Yun Lee, and Byoung-Gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2007. 3

[93] Yole Developpement. Emerging non-volatile memory technologies, 2013. 3

[94] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proc. ACM International Symposium on Computer Architecture (ISCA)*, 2009. 3

[95] Omer Zilberberg, Shlomo Weiss, and Sivan Toledo. Phase-change memory: An architectural perspective. *ACM Computing Surveys*, 45(3), 2013. 3

[96] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006. ISSN 0360-0300. 7.3