Join-based Parallel Balanced Binary Trees

Yihan Sun

CMU-CS-19-128

November, 2019

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.

Copyright © 2019 Yihan Sun

This research was sponsored by the National Science Foundation under grant numbers CCF-1314590, CCF-1833858, CCF-1629444, and CCF-1910030. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: parallel, balanced binary tree, binary tree, key-value store, map, ordered set, augmented map, index, computational geometry

To my family.

Abstract

As one of the most fundamental data structures in algorithm design and programming, *trees* play an important role in almost every area in computer science. An advantageous and effective design of trees must consider new challenges proposed by today's applications. First of all, the large scale of data requires trees to exploit parallelism and theoretical efficiency. Secondly, the comprehensiveness in applications necessitates a wide range of functions for trees. Lastly, each individual application using trees demonstrates its particularity. As a result, some generic frameworks on trees is useful in enabling simplicity and reusability for algorithms and code.

To tackle with these challenges, this thesis studies balanced binary trees in the parallel setting. This thesis proposes novel algorithms, frameworks, implementation techniques, and libraries, that are simple and efficient for a variety of large-scale applications. The core of this thesis is an algorithmic framework called the *join*-based algorithms, which bases all tree algorithms and functionalities on a single primitive *join*. The *join* function captures the essence for rebalancing, augmentation, and persistence (multi-versioning). Based on *join*, a variety of simple and efficient tree algorithms work generically across multiple balancing schemes, across multiple abstract augmentations, as well as for both in-place and persistent updates.

Theoretically, this thesis proposes a wide range of parallel tree algorithms. They all have optimal work and poly-logarithmic span. By placing conditions on the *join* function, we abstract out the preferred properties of a balancing scheme that ensures the optimal cost bounds of the *join*-based algorithms. This also leads to theoretically efficient parallel solutions to some studied applications, such as the geometry data structures and the sweepline paradigm.

In practice, the thesis work leads to an implementation of the *join*-based parallel trees, called P-Trees, in a C++ library called PAM. The library supports a complete interface for sequences, ordered sets, ordered maps, and augmented maps (formally defined in this thesis). Applying the library leads to high-performance implementation of a variety of real-world applications, such as 2D range-based searches and hybrid transactional and analytical processing (HTAP) database management systems.

P-Trees enable concise and high-performance implementations of all tested applications. Experiments show that P-Trees achieve speedups ranging from 40 to 100 across different applications on 72 cores with 2-way hyper-threading. P-Trees's performance can be up to magnitudes better than existing solutions specific to these applications, in addition to being more concise.

Acknowledgement

I generally had a very enjoyable and happy life in graduate school. I would like to thank all the people that made my Ph.D. life unforgettable, without whom the thesis would not have been possible.

I would like to express my deepest appreciation to my wonderful advisor, Guy Blelloch. When I started as a graduate student, I had very little background and knowledge about parallel algorithms. Guy is always a patient teacher and a wise advisor. I feel lucky that the very first project that I studied with Guy worked out as a conference paper, and more importantly, the series of work starting from that paper forms this thesis. Through the process, I enjoyed the great gratification of designing algorithms, proving theorems, and writing high-performance codes. Guy is obviously a great practitioner and advisor in all three aspects. He is a great role model of being a researcher, a teacher, and an advisor.

I would like to thank the rest of my thesis committee members. I got to know Danny Sleator from the 15-295 course, when I was a shy undergraduate student. He also provided me his kindly help in my graduate school application and writing skill test. I enjoyed the pizza in the 295 course for years. I especially believe that his knowledge about balanced binary trees makes him a great committee member of my thesis. I thank him for his advice and comments on the thesis.

I started to work with Andy Pavlo only about a year ago, at which time I had very little knowledge about the world of databases, but had the ambition to extend my trees to a database system. Andy offered his generous help, both in preparing the paper and in career development. Through his help, I get to know a new area, gaining new knowledge and publishing a paper in a venue that I never thought I could do. I thank him for being an excellent collaborator, and his helpful advice about my job application.

As indicated by his last name, Michael Goodrich is good and rich. Each of the few conversations with him enlightened me with new topics in research. His broad knowledge of parallel algorithms and data structures is always inspiring. I thank him for his advice about my research and the thesis, and his help for me as a young graduate student and as a new faculty member.

During my graduate studies, I also had the opportunity to interact with many other faculty members, both in CMU and in other schools. I took courses from Aarti Singh, Dave Andersen, Bernhard Haeupler, Phil Gibbons, Anupam Gupta, and Guy Blelloch, from whom I learned how to become a good teacher. I thank Charles Leiserson for giving me advice about my career and research when I was visiting MIT. I especially thank Julian Shun for being a great collaborator, for hosting many of my visits at MIT, and for giving me much advice in many aspects. I thank Guy Blelloch, Phil Gibbons, Mor Harchol-Balter, Gary Miller and Margaret Reid-Miller for organizing parties in their home. I thank Kanat Tangwongsan for being a great collaborator. I also would like to thank Andy Pavlo, Umut Acar, Julian Shun, Fei Fang, Yingjie Zhang, and Meng Jiang for their generous help for my job application.

I am very thankful to many of my fellow CMU students Naama Ben-David, Daniel Ferizovic, Yuanhao Wei, Daniel Ferizovic, Wan Shen Lim, Laxman Dhulipala, Charlie McGuffey, Sam

Westrick, Stefan Muller, Yong He, and Yu Zhao. I thank them for the enjoyable collaborations, valuable discussions, and their help with my practice talks.

I did not have a lot of leisure time activities during my Ph.D. life, but I really enjoyed the few times having parties with friends in Pittsburgh, and traveling with my friends outside Pittsburgh. I thank all my friends that invited me to their home or came by to my home for delicious food and wonderful parties, including Fan Yang, Fei Fang, Haoxian Chen, Linxi Zou, Junxing Wang, Kairui Zhang, Nan Lin, Wenting Shao, Shen-Chen Xu, Yan Wang, Yanzhe Yang, Yining Wang, Yixin Luo, Yong He, Yu Zhao, Zhanpeng Fang, and Zhou Su. I also thank those who have traveled with me Fan Gao, Lu Zheng, Tao Yu, Wanjian Tang, and Yanzhe Yin, Karui Zhang, Yifan Chen, Juanming Jiang, Weibing Pan, Jiale Zhu, and Tingyun Yan. It was great experience exploring the wonderful places of interest with them and enjoying rest time chatting.

I would like to express my special thank you to my parents. They bought me my first computer when I was six, played computer games with me, encouraged and supported me to learn to program when I was in elementary school. Although playing with computer programs at that time is not as popular (and promising) as today, they are always supportive and open-minded of allowing me to pursue my dreams. They cultivated my curiosity and interest in exploring and learning. I am very grateful to their love, and their support for all the decisions I made.

Finally, I am especially thankful to my beloved husband, Yan Gu. His unconditional support and love make me optimistic and confident. He was indeed the first one that introduced me to the world of parallel algorithms, and is a great mentor and role model for me in doing research. He is always a smart and responsible collaborator. He is my motivation to be brave in facing all the challenges in life, but is also my support allowing me to show my timidness and insecurity. He is the one who makes me want to be perfect, but is also the one who let me happily accept all my imperfections. I could not imagine how my life would be like without him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the most important and fundamental data structure used in algorithm design and programming is the *tree* structure. Most importantly, the symmetric order of trees (i.e., the in-order traversal) provides an elegant and efficient interface to organize *ordered* data. Although one can use other basic data structures such as arrays or linked lists to maintain ordering, trees are more efficient in maintaining dynamic datasets. In particular, any lookup, single insertion, deletion or update cost $O(\log n)$ time on trees with size n instead of linear. Usually taught in the first undergraduate algorithm course, algorithms and the concept of the tree data structure is the foundation of many advanced algorithms and data structures, systems, data types, applications, and important theoretical results. As a special case, the ordering allows for effective partitioning of space, and thus many geometry algorithms make use of trees (e.g., segment trees [52], range trees [56], kd-trees [56]). Designing efficient algorithms for trees has always been a pursuit for researchers in different areas, and are essential for both bounding the theoretical costs and improving practical performance of these applications.

Although basic algorithms on balanced binary trees are simple and fundamental (even a sophomore CS student would know them), the real-world applications (such as the above-mentioned ones) are far more involved than the textbook algorithms. As a result, efficient and practical tree data structures need to be adaptive to the requirements of the real-world application. Some special challenges proposed by today's applications on trees are that (1) the datasets are very large, which necessitate considering *parallelism* and *theoretical efficiency* in tree algorithm design; (2) the required functionality is usually comprehensive, obligating trees to support a *wide range of functions*; and (3) each application has its particularity, which requires *generic frameworks* on trees unifying multiple settings. *The goal of this thesis is to understand trees in the context of parallelism, and to design and develop simple and efficient algorithms, frameworks and implementations for tree data structures both in theory and for a variety of real-world applications.*

This thesis focuses on *balanced binary* [1] *trees*. Binary means that the tree can be either empty (a *nil-node*) or some data u and with two trees (noted as the *left* and *right* children) denoted as $node(T_L, u, T_R)$. Balanced means that tree height is bounded. Because many operations on trees have cost proportional to the height of the tree, it is necessary to organize binary trees in a "nearly" balanced manner. The height of a binary tree with n keys has a lower bound of $\log_2(n+1)$, and can degenerate to n in the worst case. As a result, many *balancing schemes*[2], such as AVL trees [12], red-black trees [42], weight-balanced trees [217], treaps [246], splay trees [254], etc., are designed for bounding the height of trees. In general, they all bound the tree height by $O(\log n)$ (w.h.p.[3] for treaps, amortized for splay trees).

This thesis answers the above challenges and considerations by designing a class of balanced binary trees. The proposed approaches include novel *algorithms*, *frameworks*, *implementation techniques*, and *libraries*, that are simple and efficient for a variety of large-scale *applications*, with special consideration of the *multi-core shared-memory parallel* systems. The scope of this thesis includes both theoretical and experimental studies. This thesis, in particular, proposes the *P-Tree* structure, or P-Trees, that achieve all the above-mentioned useful properties. The thesis work also includes an implementation of P-Trees in a parallel C++ library called *Parallel Augmented Maps (PAM)*. The methodology in this thesis applies to at least four commonly-used balancing schemes: AVL trees [12], red-black (RB) trees [42], weight-balance (WB) trees [217] and treaps [246].

The core of the methodology in this thesis is an algorithmic framework called *join*-based algorithms, which bases all the algorithms on top of a single primitive *join*. The function *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 $node(T_L, e, T_R)$, but satisfies the balancing criteria. We call the middle entry e the *pivot* of the *join*. An illustration of the *join* function is presented in Figure 1.1(a). This thesis uses *join* as the only primitive for connecting and rebalancing. Since rebalancing involves settling the balancing invariants, the *join* algorithm is specific to each balancing scheme.

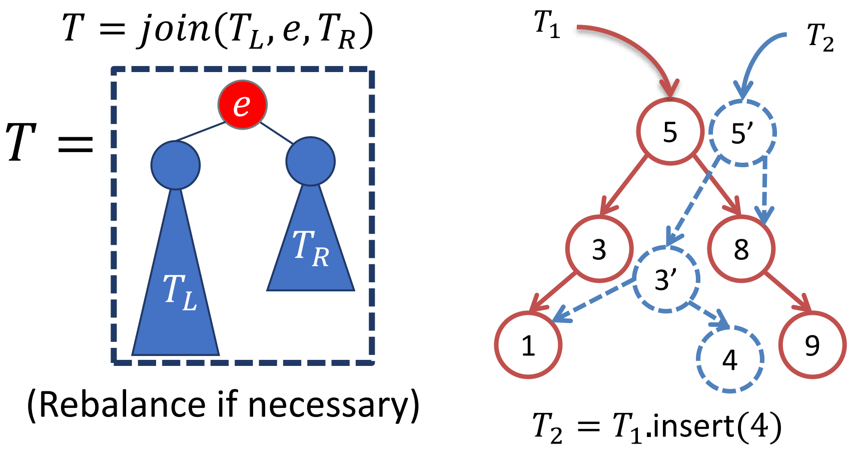
The highlight of *join*-based algorithms is that *join* captures and isolates many important functionalities and properties of trees, including rebalancing, augmentation, and persistence. More details are presented as follows.

Rebalancing. One important observation of the *join*-based algorithm is that *join* captures all balancing criteria of each balancing scheme. As such, all tree algorithms except *join*

[1] Note that similar methodology can be extended more generally, but this thesis focuses on the binary case. Our experiments also shows that our implementation outperforms many state-of-the-art tree structures, both binary and non-binary.

[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.

[3] Here w.h.p. means that height $O(c \log n)$ with probability at least $1 - 1/n^c$ (c is a constant).



(a) The *join* function

(b) A persistent insertion using Path-copying

Figure 1.1: The *join* function and a persistent insertion - Figure 1.1(a): the illustration of the *join* function. Figure 1.1(b): the illustration of a persistent insertion using path-copying.

itself can be implemented generically across balancing schemes. These algorithms need not be aware of the operated balancing scheme, but just rely on the corresponding *join* algorithm to do rebalancing. These *join*-based algorithms range from simple insertions and deletions, to more involved bulk operations. The generality is not at the cost of efficiency. In fact, all the *join*-based algorithms are still optimal in sequential work, and most of them can be parallelized with poly-logarithmic span (parallel dependency chain). These include some non-trivial and interesting theoretical results. For example, the *join*-based *union* algorithm, which combines two balanced binary search trees into one search tree (keeping the ordering), costs $O\left(m\log\left(\frac{n}{m}+1\right)\right)$ work and $O(\log n\log m)$ span on two trees with sizes m and $n \geq m$. This work bound is optimal under the comparison model. This is the first ordered set merging algorithm that achieves deterministic optimal work and poly-logarithmic span.

To show the cost bound of the *join*-based algorithms, one must consider multiple balancing schemes. Fortunately, *join* also unifies the theoretical analysis. The key idea is to define a *rank* function for each balancing scheme, that maps every tree into a real number, called its *rank*. The rank can be thought, roughly, as the abstract “height” of a tree. For example, for AVL trees the rank is simply the height of the tree, and for weight-balance trees, it is the log of the size. We then define a set of rules about rank and the *join* algorithm, that ensures a balancing scheme to be *joinable*. These rules apply to at least four existing balancing schemes. Based on these rules we are able to show generic algorithms and analysis for all joinable trees. Our bounds for *join*-based algorithms then hold for any balancing scheme for which these properties hold.

In Chapter 3, this thesis introduces all the *join*-based algorithms in detail, including the analysis of cost bounds.

Augmentation. Another important functionality that *join* enables on trees is the *augmentation*. The augmentation discussed in this thesis is to answer fast range sum queries, e.g., reporting the "sum" of values in a certain key range. This "sum" is abstract that can be defined based on any associative functions, such as taking addition, maximum, union, etc. This thesis formally defines an augmentation framework for a class of augmented trees, which maintain the partial sums of the subtree in each tree node. We call this abstract sum the *augmented value* of the subtree. This augmentation framework defines an augmentation using two functions: a *base* function g that converts a single data entry into the augmented value, and a binary *combine* function f which combines multiple augmented values. When solving a certain problem using augmented trees, the user specifies the two functions in advance (based on the application). As long as the two functions are properly defined, the *join*-based algorithms and the P-Trees in PAM directly provide simple and efficient solutions to the problem, for both theoretical analysis, and an implementation based on PAM. Most applications discussed in this thesis only requires around 100 lines of code based on the PAM library.

This thesis further defines the ordered map (key-value store) maintained by such an augmented tree as the *augmented map*, which I believe is of independent interest. This augmented map framework appropriately models many real-world applications.

It is worth mentioning that all the maintenance of augmented values in the tree happens in *join*. As a result, all the other algorithms are also oblivious to all details about augmentation. Chapter 4 will present details of augmented trees and augmented maps. Part II shows multiple applications that make use of the augmentation framework.

Persistence. This thesis also uses *join* as the only primitive to make trees *persistent*, which means that any update on trees yields a new version while preserving the old version. Persistence is useful to make the data structure purely functional, preserve history versions, and enable concurrency control on the tree. These functionalities are important in several applications, such as the snapshot isolation in database systems [183, 214, 233, 248, 273], the sweepline algorithms in computational geometry algorithms [242], transactional systems [50], etc. In particular, this thesis enables persistence using *path-copying*. Instead of copying the whole old version to preserve it, *path-copying* copies the affected path on the tree of the update. Figure 1.1(b) presents an illustration of an insertion on trees using path-copying. The algorithm copies all nodes on the path to the added node, which is at most $O(\log n)$ of them. The copied root of the new tree will then represent the entry point of T_2, and the original root node still represents the input T_1. A large portion of the two trees is shared, making the algorithm space-efficient. Similar ideas apply to all *join*-based algorithms. Another advantage of path-copying is that any concurrency is safe on P-Trees since they are purely functional. Also, path-copying allows for multiple operations on a snapshot to be visible atomically in a lock-free manner, which is more complicated to achieve using other approaches.

Interestingly, for all *join*-based tree algorithms, all such copying operations happen only in *join* and *join*'s pivots. In other words, as long as the *join* algorithm itself is persistent, there is no explicit effort needed to make the other algorithms persistent. The other algorithms are all oblivious to persistence. Chapter 5 will introduce how to realize persistence using *join* by path-copying, and how to do garbage collection based on it.

Implementations, applications and experiments. Based on the above techniques, this thesis uses *join* as the only primitive to unify different balancing schemes, enable generic augmentation, and support persistence. This thesis defines such balanced binary trees using *join*-based algorithms as *P-Trees*, and implement them in a parallel C++ library called PAM. The library effectively provides the interface for four abstract data types (ADT), which are sequences, ordered sets, ordered maps. and augmented maps. Chapter 6 will introduce the implementation details and the interface of the library.

In addition, our framework and P-Trees fit a variety of applications, achieving efficient solutions both with theoretical guarantee and high performance. As long as an application can be appropriately modeled by the framework, P-Trees directly provide support both for theory and in practice. These applications include range sum queries, hybrid analytical and transactional processing (HTAP) database management systems (DBMS), geometric queries (including 1D stabbing queries, 2D range/segment/rectangle queries), inverted index searching, transactional systems, etc. The interface and framework of PAM also allow for concise implementations of these applications. While existing implementations usually involve thousands of lines of code for each application, most of our implementation only requires about 100 lines of code. In Part II, this thesis will present several of the applications using P-Trees in details, and show the corresponding experimental results.

This thesis conducts a thorough experimental study on the parallel and sequential tree algorithms on P-Trees, as well as all the applications based on P-Trees. We compare our implementation to previous data structures, libraries, and implementations specific to each studied applications, such as database systems, range searches, etc. Experiments show that the algorithms in PAM are highly-parallelized, achieving 40-100× speedup on 72 cores with hyperthreading. They outperform or are competitive to existing sequential and parallel libraries. For applications, our implementation based on PAM outperforms almost all baseline algorithms and systems that are specific to those applications, both sequentially and in parallel.

Using P-Trees for geometric problems are up to 2.5× faster than existing solutions sequentially. On a mixed workload with concurrent updates and queries, P-Trees achieves 4-9× faster query throughput than state-of-the-art DBMSs, and is competitive in updates. With reasonable latency allowed, P-Trees can be up to 4× faster than exiting concurrent data structures on various workloads of concurrent reads and writes with different read-write ratios.

The evaluations show that the good performance of P-Trees greatly benefits from better scalability. Some of our application-specific optimizations based on P-Trees, e.g., index nesting in DBMSs, are shown to be effective in improving performance.

In the following, this thesis will present brief introductions about the methodologies adopted by this thesis, including the *join*-based algorithms, augmenting trees, and persistence using path-copying. Then we will overview the PAM library and its interface, and present a quick introduction of the applications. Each section will conclude with the relevant contributions of the thesis. Finally, this thesis discusses why *join*-based algorithms on trees are favorable and useful for modern applications, and concludes with the thesis statement.

1.1 The Join-based Algorithmic Framework

This thesis, in particular, proposes an algorithmic framework of parallel algorithms on balanced binary trees based on a single function *join*. By just plugging in a *join* function for each balancing schemes, a variety of tree operations, including *union*, *intersection*, *difference*, *filter*, *range*, *build*, *insert*, *delete*, can be implemented independently of balancing schemes. This generality is not at the cost of theoretical and practical efficiency. In the work-span model (will be defined in Section 2.1), all the *join*-based algorithms have optimal work (sequential time complexity) and poly-logarithmic span (parallel dependency chain). We list the cost of the core *join*-based algorithms in Table 1.1. The work-span model used in this thesis is described in Section 14.1.

Function	***Work***	***Span***
insert, *delete*, *update*, *find*, *first*, *last*, *range*, *split*, *join2*, *previous*, *next*, *rank*, *select*, *up_to*, *down_to*	$O(\log n)$	$O(\log n)$
union, *intersection*, *difference*	$O\left(m\log\left(\frac{n}{m}+1\right)\right)$	$O(\log n \log m)$
map, *reduce*, *map_reduce*, *to_array*	$O(n)$	$O(\log n)$
build, *filter*	$O(n)$	$O(\log^2 n)$

Table 1.1: The core *join*-based algorithms and their asymptotic costs – The cost is given under the assumption that all parameter functions take constant time to return. For functions with two input trees (*union*, *intersection* and *difference*), n is the size of the larger input, and m of the smaller.

Supporting efficient algorithms for basic operations on trees, such as insertion and deletion, are straight-forward, and were studied previously for each individual balancing scheme. The highlight of the *join*-based algorithms is that our framework presents a generic version for all balancing schemes without sacrificing the asymptotical cost. There are also some interesting functions on which supporting efficient algorithms is non-trivial, such as the set-set functions, e.g., *union*, *intersection*, and *difference*. The lower bound for comparison-based algorithms for union, intersection and difference for inputs of size n and $m \leq n$, and returning an ordered structure[4], is $\log_2 \binom{m+n}{n} = \Theta\big(m \log\big(\frac{n}{m} + 1\big)\big)$ ($\binom{m+n}{n}$ is the number of possible ways n keys can be interleaved with m keys). The bound is interesting since it shows that merging two sets (or removing a set from another) requires time proportional to the smaller sizes, instead of the larger one. Sequentially, this bound was matched by several previous algorithms such as Brown and Tarjan's sequential algorithm based on red-black trees [91]. In the parallel setting, the only algorithm before this work that achieves optimal work and poly-logarithmic depth is Blelloch Reid-Miller's algorithm on treaps [71], but the bound is in expectation. This thesis shows the first algorithm that achieves deterministic optimal work and poly-logarithmic depth. Our algorithm is based on Adams' algorithm proposed in 1992 [10, 11], which also uses *join*, but is sequential and only for weight-balanced trees.

Proving the work-optimality of the algorithms, however, is non-trivial. It is even more complicated since multiple balancing schemes can be plugged into the algorithms. Besides enabling generality of algorithm design and implementation, this thesis also unifies the theoretical analysis of different balancing schemes in the framework. The key idea is that for each balancing scheme, we define a value *rank* to capture the "height" of the given tree. This rank is used to reflect the balancing criteria of the tree. For example, as previously mentioned, an AVL tree, which is height-balanced, has rank proportional to its height. Similarly, a weight-balanced tree defines its rank based on the size. The key to proving the work-optimality of the algorithms lies in that the rank and *join* algorithms have to satisfy a few common rules that make the trees *joinable*. These rules abstract the preferred properties that make *join*-based algorithms theoretically efficient. As a result, the theoretical analysis in this thesis for all algorithms are generic across all joinable balancing schemes.

Besides theoretical efficiency, all the algorithms are also fairly simple. Figure 1.2 presents several *join*-based algorithms. They all call *join* as the primitive for rebalancing. *split* and *join2* are two other helper functions used in the code. *split*(T, k) splits a tree T by a key k, which returns two trees corresponding to the two parts of T, and a boolean value indicating if $k \in T$. *join2* (T_1, T_2) is defined similarly to *join*, but without the middle key. All the five algorithms use no more than ten lines of code. For example, *insert* just recursively inserts the key into the appropriate subtree, and calls *join* to connect the two subtrees back with the root. After inserting into one subtree, imbalance may occur, but *join* will be responsible for returning a valid balanced binary tree, guaranteeing a balanced output for *insert*. The *insert* algorithm need not deal with rotations and thus applies to all balancing schemes.

All the bulk algorithms can also run in parallel and have logarithmic depth. For example, in the *union* algorithm, the algorithm first splits one tree with the root of the

[4] By "ordered structure" we mean any data structure that can output elements in sorted order without any further comparisons—e.g., a sorted array, or a binary search tree.

other tree, such that the problem is broken up into two subproblems with smaller size. Then the two subproblems can be solved in parallel. The parallelism comes from divide-and-conquer.

The *join* function and corresponding algorithms were previously studied specifically for each balancing scheme. Adams describes a version of set-set algorithms using *join* on weight-balanced trees[5], which inspires our version of *join*-based set algorithms. Sleator and Tarjan describe an algorithm for *join* based on splay trees which runs in amortized logarithmic time [254]. Tarjan describes a version for red-black tree that runs in worst-case logarithmic time [265]. This thesis unifies multiple balancing schemes, and describes other algorithms using just *join*.

Insertion

```
insert($T, e$) {
  if $T = \emptyset$ then return singleton($e$);
  $\langle L, e', R \rangle$ = expose($T$);
  if $k(e) = k(e')$ then return T;
  if $k(e) < k(e')$ then
    return join(insert($L, e$), $e'$, $R$);
  return join($L, e'$, insert($R, e$)); } }
```

Deletion

```
delete($T, k$) {
  if $T = \emptyset$ then return $\emptyset$;
  $\langle L, e', R \rangle$ = expose($T$);
  if $k = k(e')$ then return join2($L, R$);
  if $k < k(e')$ then
    return join(delete($L, k$), $e'$, $R$);
  return join($L, e'$, delete($R, k$)); }
```

Union

```
union($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then return $T_2$;
  if $T_2 = \emptyset$ then return $T_1$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = union($L_1$,$L_2$) ||
        $T_r$ = union($R_1$,$R_2$);
  return join($T_l$,$k_2$,$T_r$);
}
```

Intersection

```
intersect($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then return $\emptyset$;
  if $T_2 = \emptyset$ then return $\emptyset$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = intersect($L_1$,$L_2$) ||
    $T_r$ = intersect($R_1$,$R_2$);
  if $b$ then return join($T_l$,$k_2$,$T_r$);
  else return join2($T_l$,$T_r$); }
```

Difference

```
difference($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then $\emptyset$;
  if $T_2 = \emptyset$ then $T_1$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = difference($L_1$,$L_2$) ||
    $T_r$ = difference($R_1$,$R_2$);
  return join2($T_l$,$T_r$);
}
```

Filter

```
filter($T$,$f$) {
  if $T = \emptyset$ then return $\emptyset$;
  ($L$,$e$,$R$) = expose($T$);
  $L'$ = filter($L$,$f$) || $R'$ = filter($R$,$f$);
  if $f(e)$ then return join($L'$,$e$,$R'$);
  else join2($L'$,$R'$); }
```

Map and Reduce

```
map_reduce($T, g', f', I'$) {
  if $T = \emptyset$ then return $I'$;
  $\langle L, k, v, R \rangle$ = expose($T$);
  $L'$ = MapReduce($L, g', f', I'$) ||
        $R'$ = MapReduce($R, g', f', I'$);
  return $f'(L', f'(g'(k, v), R'))$; }
```

Figure 1.2: Pseudocode of some *join*-based functions – The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

Contributions. The contributions of this thesis related to *join*-based algorithms include the following.

[5] Adams' version had some bugs in maintaining the balance, but these were later fixed [159, 257].

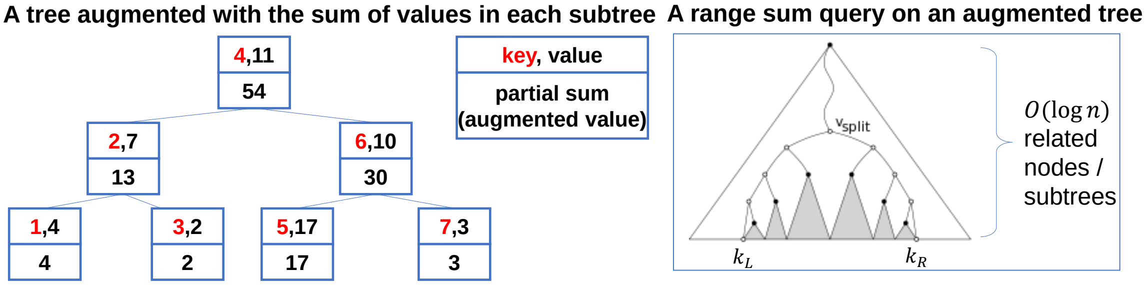
1. The *join*-based algorithm framework on balanced binary trees, which unifies different balancing schemes, including (but not limited to) AVL trees, red-black trees, weight-balanced trees and treaps. The framework abstracts out rebalancing for other tree algorithms in *join*. I believe the framework is general and can be applicable to even more balancing schemes in future work.
2. The definition of joinable trees, which is a set of rules. A balancing scheme fits in the *join*-based algorithm and analysis as long as it satisfies the rules.
3. A list of parallel and sequential algorithms on trees including *union*, *intersection*, *difference*, *build*, *insert*, *delete*, *range*, *filter*, *map_reduce*, etc., which are all simple, work-optimal and have polylog depth.
4. The first proof of the work-optimality of Adams' set algorithms as well as its extension to other balancing schemes.
5. An experimental study of the *join*-based algorithms and comparison to existing implementations.

Chapter 3 will introduce the *join* algorithm on each balancing scheme. The chapter will then present the *join*-based algorithmic framework, a variety of *join*-based algorithms (many of them in parallel), as well as the theoretical analysis of the work and span bound of these algorithms.

1.2 Augmented Trees and Augmented Maps

Real-world problems are usually complicated and require much more functionality than a plain tree could support. In practice, instead of inventing brand-new data structures, it is common just to augment trees to adapt to specific requirements of certain applications [118]. As such, certain queries can be more efficient by taking advantage of the augmentation on trees. This thesis proposes and formalizes an augmentation framework on balanced binary trees.

Augmentation on trees is a wide concept and is applied in many previous results. For example, just augmenting each tree node with the size of its subtree can make many useful queries more efficient, such as selecting an element with a certain index in a tree. Another quick example is to answer queries about the "sum" of values in a certain key range. As an example of such a *range sum*, consider a database of sales receipts keeping the value of each sale ordered by the time of sale. When analyzing the data for certain trends, it is likely useful to quickly query the sum or maximum of sales during a period of time. Then storing partial sums in each subtree is beneficial because any such key range corresponds to at most $O(\log n)$ such partial sums. More generally, the augmentation can be used for quick interval queries, high dimensional range queries, inverted indices (all described later in this thesis), segment intersection, windowing queries, point location, rectangle intersection, range overlaps, and many others.



A tree augmented with the sum of values in each subtree

A range sum query on an augmented tree

Figure 1.3: Augmented trees and range sum queries on augmented trees – An illustration of a tree augmented with the sum of all values in each subtree and a range sum query on an augmented tree.

In particular, this thesis formalizes a class of *augmented trees*, with respect to the data entries as *key-value pairs*. For each tree (or subtree), an *augmented value* is assigned as a partial sum of all entries in it. This partial sum can be based on any associative operations, e.g., addition, max, or union. As such, to define an augmented tree, besides the parameters of the key type and the value type, two augmenting functions are required: a *base function* g which gives the augmented value of a single element, and a *combine function* f (must be associative) which combines multiple augmented values, giving the "sum" (augmented value) of the subtree (see more details in Chapter 4). These functions are chosen ahead of time, based on the applications. For the same dataset, different augmentations (i.e., different augmenting functions) can lead to different functionality. In the sales receipts example, besides using the sum of sales as the augmented value to report sale sum in a range, the sales above a threshold can also be reported in $O(k\log(n/k+1))$ (k is the output size) if the augmentation is the maximum of sales, or in $O(k+\log n)$ time [203] with a more complicated augmentation.

Such augmented trees can be implemented based on the *join*-based algorithms. Because of the associativity of the combine function, the augmented value in a node can be directly computed using the entry in the node itself and the augmented values of its two children. As a result, all update of the augmented value in a tree node happens only when a node is assigned a new value or its children get updated. Outside *join*, such scenario happens only on the pivotof a *join* algorithm. As a result, to achieve augmentation on P-Trees, only the *join* algorithm need to be "augmented". All the other algorithms can be oblivious to the augmentation and need not be changed or re-implemented. Experiments show that the additional cost for maintaining the augmentation is reasonably small (typically around 10% for simple augmenting functions such as summing the values or taking the maximum).

Such augmentation on trees is useful for answering range-related queries because of the partial sums maintained in the tree. Experiments show that with a simple augmentation, such as the addition, the range sum queries can be answered almost as fast as a *find* operation. It is worth noting that because of the abstract framework on arbitrary augmentation, even high-dimensional range queries (e.g., the augmented values themselves are trees) can be implemented based on P-Trees with no more difficulty than a simple range sum. They both only requires to properly define the base and combine functions.

Augmented Maps. The augmented trees as defined above represent a special type of ordered maps (also called key-value store, dictionary, table, or associative array) associated with the augmentation functions. The ordered map is very important in practice as is indicated by many large-scale data analysis systems such as F1 [250], Flurry [25], RocksDB [239], Oracle NoSQL [221], LevelDB [191]. This thesis further define this type of ordered map as *augmented maps*. This means to also assign an augmented value using the base and combine function on an ordered map. This augmented map framework is a more general concept than augmented trees in modeling a collection of different real-world applications. In fact, many seemingly-unrelated applications can be modeled by the framework by just plugging in the keys, values, and augmented functions. Section 1.5 will overview more details about these applications, such as some geometric queries and inverted indexes searching.

The augmented map, as an abstract data type (ADT), is independent of the data structure implementation. In fact, using other data structures to support augmented maps can provide different solutions to certain applications. For example, the prefix structures, which is a data structure proposed in this thesis, maintain augmented maps by storing the augmented values of all prefixes (i.e., the prefix sums) of a map. When applied to geometric problems, such data structures provide solutions similar to sweepline algorithms [259].

As long as a problem can be formalized as augmented maps, by directly plugging in P-Trees and prefix structures, one can achieve two different solutions, which can be appropriate in different settings and answering different queries.

Contributions. The contributions of this thesis on the side of augmentation include the following.

1. An augmentation formalization framework for *augmented trees*.
2. The definition and interface of *augmented maps* and the study of its applications on various areas and problems.
3. The definition of the data structure *prefix structures* and the parallel algorithms for them.
4. The implementations of the augmented map algorithms using both *join*-based augmented trees and prefix structures.
5. An experimental study of augmented tree algorithms using *join*, and the implementation of augmented maps for various applications.

Chapter 4 presents algorithms for supporting augmentation on trees, the formal definition of augmented maps, and the prefix structures. Several applications of augmentation are presented in Part II.

1.3 Persistence and Multi-versioning

A *persistent* data structure means that any update on the data structure does not destroy the previous version but yield a new version. The concept was proposed since the early days of Lisp (1960s), and is also employed in maintaining some previous multi-version data structures [43, 127, 255]. This property is useful in many scenarios. For example, in a class of sweepline algorithms [249], the algorithm updates a data structure on each event point. For query purposes, the data structure needs to keep all history versions of itself, which can be implemented by a persistent data structure [242]. Another example is the multi-version concurrency control (MVCC) [61, 181, 214, 226, 229, 237] in database systems with concurrent updates and queries. MVCC aims at letting updates generate new versions while preserving the old version for ongoing queries. This techniques guarantees queries to work on a consistent version. MVCC also improves the throughput of the system because queries and updates do not need to wait for each other or coordinate with each other by locks.

There are multiple ways to enable persistence on trees. For example, one can copy the whole old version, or at least a local part (e.g., a subtree) of the update to preserve the previous version. This approach usually results in unnecessary copying, which is not space-efficient. However, this makes each version isolated, which simplifies queries. Persistence can also be implemented by the version lists (or version chains) [60, 61, 181, 226, 237, 273], which maintains a list of all history versions for each object (i.e., a tree node) along with the timestamps. Then a search query also comes with a timestamp, and will check the visibility of each version. Although extra metadata needed, any single update only results in one extra data entry, which is space efficient. The drawback of such an approach is that queries may be slowed down because of checking the long list. Another approach is based on path-copying [242], which means to copy the affected path of the update. As illustrated in Figure 1.1(b), although most of the parts of the tree are shared, a copied path can distinguish the old and the new version. By copying a logarithmic number of tree nodes, path-copying also allows for queries to grab an “isolated” current version by reading just the root pointer. This is a reasonable tradeoff between space and time. There are also other solutions for persistence using a combination of the three.

This thesis uses path-copying to enable persistence on balanced binary trees. In particular, this thesis uses *join*-based algorithm for path-copying. The observation is that as long as the *join* algorithm itself is persistent, the only extra copying in the algorithms are just the pivots of the *join* algorithms. This again allows for a very simple implementation—just like the rebalance and augmentation, all the *join*-based algorithms need not be aware of persistence, but can rely on *join* to deal with this. In addition, enabling persistence does not increase the asymptotical cost of the *join*-based algorithms, and the extra space is no more than the time complexity.

Using path-copying has several advantages in the studied applications in this thesis. First of all, as mentioned, each version is "isolated" as a snapshot such that grabbing a version is fast and simple. Secondly, a version is visible immediately after the root is accessible, and can be invisible before that. This allows multiple updates to be visible atomically. For example, some the bulk *join*-based algorithms, such as *union*, can make a set of new elements inserted into a tree atomically. Finally, path-copying makes the tree *purely functional.* The algorithms never destroy the input. This allows for a high-level programming-friendly interface of the trees.

On the other hand, path-copying results in a number of copied nodes, which need be collected once they become out-of-date. The essence that multiple versions share tree nodes complicates garbage collection (GC) on such path-copying data structures. This thesis adopts reference counting garbage collector, which means to record the number of references to each tree node. Once a version is collected, the GC algorithm decrements the counter of related tree nodes, and frees all whose counter reaches zero.

Contribution. The contribution of this thesis regarding persistence is summarized as follows.

1. Path-copying algorithms using *join* for making tree algorithms (especially parallel algorithms) persistence (or purely functional) and the implementation.
2. Although the general idea of path-copying has a long history, this thesis first extends it to the parallel bulk operations.
3. Efficient and effective garbage collection algorithms for path-copying and the implementation of the GC algorithm.

Chapter 5 will present details about using *join*-based algorithms to achieve persistence by path-copying. The garbage collection algorithm to reclaim tree nodes is also presented in the same chapter.

1.4 The PAM Library and the Interface

With all the above-mentioned properties, including parallelism, *join*-based algorithms, augmentation and persistence, this thesis defines such balanced binary trees as P-Trees. P-Trees are further implemented in a parallel library PAM (Parallel Augmented Maps). The interface provided by PAM is based on four abstract data types, including sequences, ordered sets, maps and augmented maps [6]. When creating a specific data type, the user specifies the parameters (all or part of the key type K, the comparison function $<$ on K, the value type V and the augmentation type *aug*) based on the applications, then a variety of functions are supported by the interface. The data types can be summarized as follows.

[6] They are somehow almost all datatypes one would like to support by binary 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 *build*, *append*, *select* (pick up the i-th element), *split_at* (split at the i-th element), etc.
2. **`Ordered_set(K,`$<_K$`)`**. On top of the sequence interface, if K has a total ordering, which associates K a comparison function $<_K : K \times K \mapsto Bool$, then an ordered *set* implementation can be formed. This makes the tree a *binary search tree* (BST) from this point. Beyond the sequence functions, many aggregate functions for sets (e.g., *union*, *intersection*, *difference*, etc.) are then added to the interface. Even the insertion and deletion can be performed 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[7]. This augmentation structure is chosen based on the specific desired applications, and consists 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 4.4. 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.

In addition to the wide range of functions supported, the library is also theoretically-efficient and highly parallelized. All algorithms are work-efficient meaning that their theoretical work bound asymptotically no more than the best known sequential algorithms. All parallel algorithms have polylog depth. Experiments show that all algorithms in the library achieve good speedup and scalability in practice. The library also support generic user-specified augmentations. The interface for augmentation is concise, and the library directly gives an efficient underlying implementation. Also, the tree structure in PAM is persistent, implemented by path-copying. Experiments show that the extra cost of both maintaining augmentation and persistence is small. Because of path-copying, PAM also supports multi-versioning, which makes it safe for concurrency. Each concurrent process

[7] Note that as a data type, the augmented map abstraction itself does not depend on augmented trees. See Chapter 4.4 for more details.

	n	**m**	**T_1**	**T_{144}**	**Spd.**
PAM (with augmentation and persistence)					
Union	10^8	10^8	12.517	0.2369	52.8
Union	10^8	10^5	0.257	0.0046	55.9
Find	10^8	10^8	113.941	1.1923	95.6
Insert	10^8	–	205.970	–	–
Build	10^8	–	16.089	0.3232	49.8
Build	$\mathbf{10^{10}}$	–	**1844.38**	**28.24**	**65.3**
Filter	10^8	–	4.578	0.0804	56.9
Multi-Insert	10^8	10^8	23.797	0.4528	52.6
Multi-Insert	10^8	10^5	0.407	0.0071	57.3
Range	10^8	10^8	44.995	0.8033	56.0
AugLeft	10^8	10^8	106.096	1.2133	87.4
AugRange	10^8	10^8	193.229	2.1966	88.0
AugRange	$\mathbf{10^{10}}$	$\mathbf{10^8}$	**271.09**	**3.04**	**89.2**

Table 1.2: Timings in seconds for various functions in PAM – "T_{144}" means on all 72 cores with hyperthreads (i.e., 144 threads), and "T_1" means the same algorithm running on one thread. "Spd." means the speedup (i.e., T_1/T_{144}). For insertion we test the total time of n insertions in turn starting from an empty tree. n is the size of the operand tree. m for *find*, *range*, *aug_left* and *aug_range* means the number of invocations of these functions. m for *union* and *multi_insert* means the size of the other operand.

can access the same version of a tree and get a snapshot of it. Then each process can modify their local copy without affecting or being affected by other processes.

The top-most level of the interface, which is the augmented map, is an abstraction fitting many real applications. The augmented map abstraction and the PAM interface greatly simplify the implementation of these applications. Most applications mentioned in this thesis, 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. Some results are shown in Table 7.1. 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.

Contributions. The contribution of this thesis regarding the library is as follows.

1. A parallel C++ library PAM implementing P-Trees, supporting full-featured interfaces for sequences, ordered sets, ordered maps and augmented maps using *join*-based algorithms. The library is highly-parallelized, safe for concurrency, theoretically-efficient, persistent (purely functional), supporting general augmentations, supporting MVCC with effective garbage collection, and applicable to four balancing schemes. The library is open source [261].
2. An implementation of prefix structures for augmented maps.

3. An experimental study of all set and map algorithms in PAM.

Chapter 6 will introduce the library, including the functions supported, the user interface and some implementation details. The experimental results are demonstrated in each application in Part II.

1.5 Applications

The balanced binary trees are widely-used for a variety of applications. Plugging P-Trees in, we can obtain interesting new results and solutions to these applications. Furthermore, using the PAM library can also provide efficient implementations for these applications. Because of the properties of P-Trees and the PAM library, these solutions are usually efficient both in theory and in parctice.

Range-based Computational Geometry Queries. A class of applications that this thesis studies are the range-based computational geometry queries. In particular, this thesis looks at range, segment and rectangle queries in 2D Euclidean space. These queries are fundamental problems in computational geometry, with extensive applications in many domains. The queries take a list of input points, segments or rectangles, and report the relevant objects in a given range (e.g., all points in a query rectangle, all segments intersecting with a query segment, etc.). The definitions of these problems are formally stated in Chapter 2. Efficient solutions to these problems are mostly based on variants of range trees [57], segment trees [52], sweepline algorithms [249], or combinations of them.

Although there is a large body of work on sequential algorithms and data structures [56, 58, 101, 106, 130, 131], in parallel, there have been a gap between theory and practice. There exist many theoretical results on parallel algorithms and structures for such queries [17, 32, 35, 141], but efficient implementations of these structures can be complicated. The parallel implementations we know of [100, 161, 172, 205] do not have useful theoretical bounds. One major reason is the delicate design of algorithmic details required by the theoretically-efficient structures. Also, various types of such data structures, such as range trees, segment trees, sweepline data structures, are involved. They are designed differently such that implementing even one of them is difficult, not to mention implementing multiple such data structures for different queries. To overcome these challenges, this thesis develops theoretically efficient algorithms which can be implemented with ease and also run fast in practice, especially *in parallel*.

Our approach is based on the augmented maps defined in this thesis. In particular, all the three problems can be modeled as two-level map structures: an outer level map augmented with an inner map structure. For different settings and query types, this thesis develops five structures on top of the augmented map interface corresponding to different problems and queries. To implement these augmented maps, this thesis uses both the augmented tree structures, and *prefix structures*, both introduced in this thesis. By combining the five two-level map structures with the two underlying data structures

		Build		**Query**	
		Work	**Span**	**List-all**	**Count**
Range Query	**Sweepline**	$n \log n$	n^{ϵ}	$\log n + k \log\left(\frac{n}{k} + 1\right)$	$\log n$
	Tree	$n \log n$	$\log^3 n$	$\log^2 n + k$	$\log^2 n$
Segment Query	**Sweepline**	$n \log n$	n^{ϵ}	$\log n + k$	$\log n$
	Tree	$n \log n$	$\log^3 n$	$\log^2 n + k$	$\log^2 n$
Rectangle Query	**Sweepline**	$n \log n$	n^{ϵ}	$\log n + k \log\left(\frac{n}{k} + 1\right)$	$\log n$
	Tree	$n \log n$	$\log^3 n$	$\log^2 n + k \log\left(\frac{n}{k} + 1\right)$	$\log^2 n$

Table 1.3: Theoretical costs of 2D geometric queries - Bounds are asymptotical in Big-O notation. n is the input size, k the output size. $\epsilon < 1$ can be any given constant. “Sweepline” means the sweepline algorithms, “Tree” the two-level trees. “List-all” means to list all relevant objects. “Count” means to return the number of relevant objects. We note that not all query bounds are optimal, but they are off optimal by at most a $\log n$ factor.

as the outer map (the inner maps are always implemented by augmented trees), this thesis presents a total of ten different data structures for the geometric queries.

Especially, the construction algorithms of the prefix structures for the three problems share some common properties in the augmenting functions. This thesis also proved the cost bound of the parallel construction algorithm of the prefix structures for these three problems. Interestingly, the algorithms based on the prefix structures resemble the standard sweepline algorithms. Therefore, our algorithms also parallelize a family of sweepline algorithms that are efficient both in theory and practice.

All the proposed data structures are efficient in theory. We summarize the theoretical costs in Table 1.3. All the proposed data structures are also fast in practice. Our implementation achieves a 33-to-68-fold self-speedup in construction on 72 cores (144 hyperthreads), and 60-to-126-fold speedup in queries. Sequentially, our implementation is competitive or can be orders of magnitudes faster than existing implementations.

Beyond being fast, our implementation is also concise and simple. On top of PAM, each application only requires about 100 lines of C++ code even for the parallel version. We note that PAM implements general-purpose augmented maps, and does not directly provide anything special for computational geometry. For the same functionality, existing implementations use hundreds of lines of code for each sequential implementation.

HTAP Database Systems. An important application of trees is to build indexes for database management systems (DBMS). There are two major trends in modern data processing applications that make them distinct from database applications in previous decades [230]. The first is that analytical applications now require fast interactive response time to users for queries. The second is that they are noted for their continuously changing data sets. There has been research on studying specialized DBMS for online analytical processing (OLAP) which makes complicated analytics fast by some query optimizations. On the other hand, for enhancing updates, there are also research focusing on improve online transactional processing (OLTP) throughput. More recently, the concept of HTAP (hybrid of transactional and analytical processing) gains attention, aiming at achieving both analytical queries and transactional updates efficiently and correctly. This poses several challenges to the DBMSs, especially in the parallel and concurrent setting. Foremost is that queries need to analyze data that are up-to-date. Data has immense value as soon as it is created, but that value can diminish over time. Thus, it is imperative that the queries access the newest data generated, without being blocked or delayed by ongoing updates or other queries. Secondly, the DBMS must guarantee that each query has a consistent view of the database. This requires that the DBMS atomically commit transactions efficiently in a non-destructive manner (i.e., maintaining existing versions for ongoing queries). Finally, both updates and queries need to be fast, e.g., exploiting parallelism or employing specific optimizations.

Unfortunately, existing solutions of query optimizations usually suffer from overhead when dealing with dynamic data, which slows down updates. To make updates to proceed faster, on the other hand, usually requires some form of multi-version concurrency control (MVCC) [62, 211, 269]. Instead of updating tuples in-place, with MVCC each write transaction creates a new version without affecting the old one so that readers accessing old versions still get correct results. One common solution to MVCC is to use *snapshot isolation* (SI) [59], where every transaction sees only versions of tuples (the “snapshot”) that were committed at the time that it started. However, this again is difficult (or costs overhead) to be combined with many effective query optimizations and may slow down queries. More detailed explanation will be presented in Section 11.1.

To overcome these challenges and enable fast concurrent updates and queries, this thesis proposes a solution based on P-Trees. Being persistent using path-copying, P-Trees by default allows any thread to see a snapshot of the current version. As stated in Section 1.3, using path-copying has several benefits, such as making reading a snapshot fast and allowing for multiple operations to be atomic, which are all useful for DBMSs.

Additionally, because the P-Tree interface supports arbitrary key and value types, it is easy to set the value type also as a tree. This allows for *nested tree* structures for building *nested indexes*. In OLAP workloads, nested indexes can effectively represent the logical hierarchy on data. In many cases, the nested indexes provide a similar if not equivalent functionality as table denormalization and materialized view of a pre-join[8]. As such, nested indexes makes queries faster, and path-copying makes updates on such nested indexes still reasonably efficient.

Finally, the *join*-based algorithms supports high parallelism. For example, multiple updates can be committed atomically and in parallel by a *multi_insert* or *multi_delete* algorithms (they both use divide-and-conquer similar to the *union* and *difference* algo-

[8] It is worth noting that this “join” terminology is not the balanced binary tree primitive in this thesis, but is the operation in the relational database.

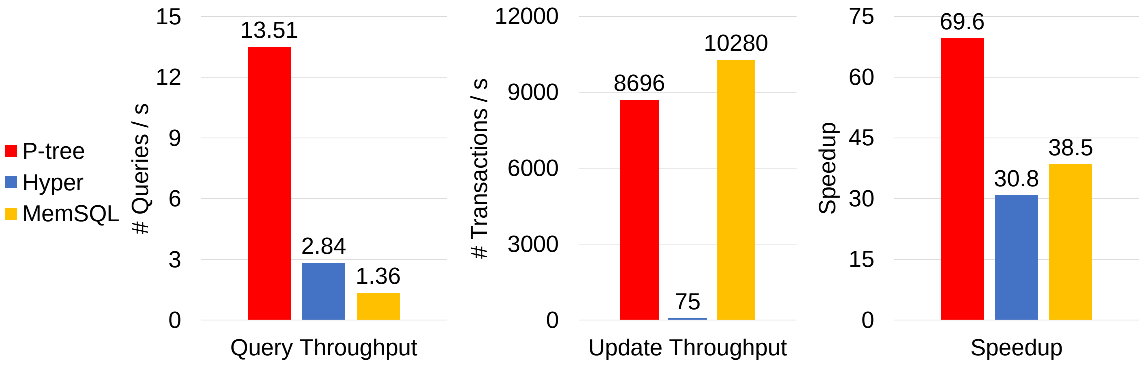


Figure 1.4: A summary of P-Trees' performance on an HTAP database system – The workload consists of concurrent analytical queries from TPC-H, and transactional updates similar to TPC-C.

rithms). Also, some other operations, such as *range*, *filter*, *map_reduce*, etc., allows to analyze on a snapshot efficiently.

Combining all these techniques, P-Trees demonstrate good performance on HTAP workloads. We tested P-Trees on an artificial HTAP benchmark which combines TPC-H [7] queries and TPC-C-style transactions [6] adapted into TPC-H workload. A summary of the result is shown in Figure 1.4. Compared to two state-of-the-art in-memory DBMSs, P-Trees is 4-9× faster in queries, and almost competitive in updates. Evaluations show that the performance gain mainly comes from good parallelism (about 70× speedup on 72 cores with hyperthreading), and the index nesting optimization (2× improvement on average).

A Data Structure for Concurrent Reads and Writes. One potential issue of path-copying-based data structures is that concurrent operations on the tree do not come into effect on the same version. In particular, any concurrent thread only updates its local snapshot. To guarantee serializability, it is essential to employ additional techniques.

For simple point updates, this thesis proposes to use *batching*. This means to only use one global writer to collect all the concurrent updates, coordinate internally, and commit altogether in parallel. Because of the parallel bulk operations supported by P-Trees using divide-and-conquer, such bulk operations (e.g., *multi_insert*) avoid conflict and contention, which is likely to be more efficient than invoking all operations concurrently. Similar approaches have also been shown to be efficient in practice in previous work [1, 18, 155, 247, 266].

Theoretically, P-Trees with batching yields a concurrent data structure allowing for lock-free (guaranteed system-wide progress) writes, such as insertion, deletion and updates, as well as wait-free (guaranteed per-thread progress) read-only queries, such as searching and range queries.

Experiments show that using the P-Tree with batching is efficient on workloads of concurrent updates and searching with different read-write ratios. We use four workloads in YCSB (Yahoo! Cloud Serving Benchmark). On all tested workloads, with reasonable latency, P-Trees outperforms state-of-the-art concurrent data structures which even do not support snapshotting.

Version Maintenance on Concurrent Systems. In a concurrent system, concurrent read-only transactions access the latest state of the system, and concurrent write transactions update the latest state of the system and commit. Such a system is useful in database management systems (DBMS) [61, 181, 214, 226] software transactional memories (STM) [229, 237], operating systems [206, 207], etc. Some solutions, such as Read-Copy-Update (RCU), makes writers wait until the current version is not used by any other threads. This is not effective when readers can be arbitrary long.

Many other solutions are based on multi-versioning, which allows readers and writers to proceed because they each work on an isolated version. Despite the efficiency in the transactions, a special consideration of such multi-versioned systems is that old versions can be kept even after new versions are generated. As a result, these old versions eventually need to be detected and collected in time, when no future queries will access it, sooner the better. The detection of such out-of-date versions is defined as the *version maintenance* (VM) problem. Ben-David et al. [50] propose a framework to VM problems. Detecting when a version is safe to collect, this framework requires an underlying functional data structure which supports appropriated garbage collection. The property that P-Trees support snapshotting using path-copying and reference counter GC makes P-Trees good candidates for a VM solution. This thesis combines P-Trees with a simple lock-free VM solution, yielding a multi-versioning transactional system that is lock-free, serializable, guaranteeing no-abort for all readers and one writer, and with safe and precise GC. P-Trees can also be combined with other VM solutions in previous work such as hazard pointers [210], epoch-based GC [111], Read-Copy-Update (RCU) [206, 207], and a wait-free safe and precise (WFPS) VM solution [50].

Other Applications. This thesis also tested other interesting applications of P-Trees. They include simple range-sum and range-max searching, interval trees for 1D stabbing query based on the augmented map interface, inverted index searching with ranked search and write-efficient set operations.

Contribution. The contribution of this thesis on the application side can be summarized as follows.

1. A simple and efficient parallel interval tree implementation using augmented maps in PAM.
2. A framework and simple and efficient parallel implementations for various 2D-range based queries. This thesis proposes approaches and implementation based on

both the multi-level P-Trees and sweepline algorithms using P-Trees with prefix structures, respectively.

3. A concurrent data structure using batching and parallel bulk algorithms in P-Trees supporting concurrent insertion, deletion, update, and various read-only operations such as lookup, select, and range searches. The system supports wait-free reads (e.g., lookup and range query) and lock-free writes (insertion, deletion and update).
4. An approach for MVCC with precise and safe garbage collection in transactional systems based on P-Trees.
5. Defining and formalizing the nested index in DBMS index building, along with an implementation of them using nested P-Trees. The index nesting can enable similar query optimizations as denormalization and pre-join, but in a more space-efficient manner and allowing for fast updates. The index nesting can greatly improve OLAP query performance.
6. An in-memory HTAP DBMS supporting TPC-H analytical queries and TPC-C-style transactions, which outperforms state-of-the-art systems.
7. Thorough experimental study on the applications based on P-Trees. This thesis compares our implementation with previous data structures, libraries, and implementations specific to each studied applications, such as database systems, range searches, etc.
8. The first experimental study of write-efficient tree-based set algorithms.

Part II introduces all applications in details, including how they fit in the framework, some application-specific optimizations, the implementation, and experimental evaluations.

1.6 Outline of This Thesis

The results in this thesis are primarily based on some previous publications, and also include new results that are unpublished. They are listed below.

- [74] (full version in [69]) Just Join for Parallel Ordered Sets. Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016. Included in this thesis in Chapter 3.
- [262] (full version in [260]) PAM: Parallel Augmented Maps. Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018. Included in this thesis in Chapters 4 to 6, 8, 9, 12, and 13.
- [259] (full version in [258]) Parallel Range, Segment and Rectangle Queries with Augmented Maps. Yihan Sun and Guy E. Blelloch. In *Algorithm Engineering and Experiments (ALENEX)*, 2019. Included in this thesis in Chapters 4 and 9.

- [263] On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multiversioned Indexes. Yihan Sun, Guy E Blelloch, Andrew Pavlo, and Wan Shen Lim. In *Proceedings of the VLDB Endowment (PVLDB)*, 2020. Included in this thesis in Chapter 11.
- [50] (full version in [48]) Multiversion Concurrency With Bounded Delay and Precise Garbage Collection. Naama Ben-David, Guy E. Blelloch, Yihan Sun, and Yuanhao Wei. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, 2019. Partially included in this thesis in Chapters 5, 10, and 12.
- [148] (full version in [149]) Algorithmic Building Blocks for Asymmetric Memories. Yan Gu, Yihan Sun, and Guy E. Blelloch. In *European Symposium on Algorithms (ESA)*, 2018. Partially included in this thesis in Chapter 14.

Many other papers [76, 77, 78, 80, 146] by the thesis author in the graduate school are not included in this thesis.

1.7 Why *join*-based Algorithms?

As stated *join*-based algorithms can enable multiple useful functionalities on trees, but why they are essential and effective for today's applications? Now this question can be better answered after overviewing the challenges, techniques and the scope of applications that are interested to this thesis. To understand the advantage of the *join*-based algorithms, recall the real-world applications and the three features of them mentioned at the beginning of this thesis, which are the large scale of the dataset, the comprehensiveness of the functionality required, and the specificity and complication of each application. They all put forward new challenges to the tree algorithm design. This section will discuss each challenge respectively and show how *join* can deal with the challenges.

The large scale of dataset to be dealt with requires higher performance of trees. For example, a database can be dealing with terabytes of data. Even sequentially reading or scanning all of them in memory requires hours, not to mention applying complicated algorithms on the data. This necessitates *parallelism*, *concurrency*, and *theoretically efficiency* in the tree algorithm design. Fortunately, the advent and popularization of parallel systems allow for much better throughput, which is capable to handle the scale of data we are facing. Especially, even a single multi-core machine, which is the main setting considered by this thesis, can have terabytes of data in memory, and hundreds of physical threads, suggesting significant potential of performance improvement. On the other hand, new theoretical models and tools for studying parallelism are gradually getting matured, which allow for rigorous theoretical analysis on predicting and bounding the worst-case performance of parallel tree algorithms. To achieve high performance on trees, one should consider optimal work (sequential time complexity) and polylogirithmic depth (parallel dependency chain), linear scalability in practice, as well as lock-freedom or wait-freedom for concurrency.

The comprehensiveness of the functionality required by a variety of applications further requires a *full-featured interface* supporting various data types and functions, as well as the ability to handle *analytical queries and dynamic updates* (and even a combination of both simultaneously) correctly and efficiently. This requirement is more than just supporting simple insertions, deletions and lookups, as most of the previous work focuses on. For example, in the geometric applications as mentioned, many queries are related to range-based searching. Similarly in database systems, instead of dealing with point updates and lookups, some bulk operations on the whole database, such as filtering, reducing, and bulk updates, can be useful. These bulk updates are likely to be more efficient than doing multiple point operations. Correctness criteria such as serializability and atomicity are also important for large and concurrent transactions. Multi-versioning is also favorable for duarbility, transaction isolation and concurrency control, which further necessitates timely and safe garbage collection.

The third feature is the specificity and complication of each application. As a result, a large number of different tree structures are proposed to achieve different specific functionalities. For example, to keep trees balance, many balancing schemes are proposed from back to the 70s. They have different properties in specific settings, maintain different balancing information, can achieve different bounds in certain problems, and thus need different algorithms in design. Another example is also the range-based geometric problems, many different tree structures are proposed for different query types, including interval tree [118], range tree [53], segment tree [55], etc. Although all based on trees and sharing some similarities (e.g., they are all space-partitioning), it was believed that their designing focus and details make their implementation fairly different, and to implement all of them, little code can be reused. The combination of different balancing schemes, algorithms, and applications complicates the situation such that usually one needs to design and implement a different tree structure for each case. Further considering the above two challenges, e.g., parallelism, theoretical analysis and multi-versioning, this greatly increases designing and coding effort. As a result, it would be nice if some useful frameworks and generic methodology can be developed on trees, both for simplifying theory analysis and for reducing the coding and debugging effort in practice.

Balanced binary trees have been studied for more than 50 years. Why are traditional algorithms on balance binary trees facing new challenges as mentioned above? One possible reason lies in that traditional algorithm design on balanced binary trees usually uses insertion and deletion as primitives, which are plagued with these new challenges. First of all, these two primitives are inherently sequential. Even though multiple insertions and deletions can be processed simultaneously, they still have to be sequentialized 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. For example, if one wants to insert multiple elements into a tree, adding them using multiple single insertions is often not a good solution because it has worse cache locality and is harder to achieve parallelism. Finally, conventional insertion and deletion algorithms on balanced binary trees are proposed for each specific balancing scheme, making implementations less generic even in the sequential setting, and much more complicated in parallel.

Instead of using insertions and deletions, using *join* as the primitive provides effective solutions to the three challenges.

First of all, *join* can easily enable the use of divide-and-conquer in algorithms, i.e., divide trees into parts, solve subproblems and *join* them back as the result. This scheme is friendly to parallelism because the subtrees in the recursive subproblems are non-overlapping, and thus can be dealt with in parallel. Our experiments show that P-Trees achieve almost linear speedup in most implemented algorithms and applications.

Secondly, *join* is especially useful for some bulk operations such as merging two trees because *join* operates two trees instead of single elements. As a result, based on *join*, it is simple to design many bulk algorithms including inserting a batch of elements, filtering by a certain predictive, extracting a key range, etc. They are all implemented in the PAM library.

Most importantly, using *join* as a primitive allows for generality on trees. This is embodied in two aspects: (1) *join* captures low-level rebalancing criteria, such that a variety of tree operations can be implemented generically, independent of balancing scheme; In fact, for each function (except *join*) in PAM, all balancing schemes share the code; and (2) *join* captures high-level functionalities of augmentation and persistence, such that a variety of real-world problems can be implemented all by the *join*-based algorithms with minimal variance; In fact, all applications in this thesis can be implemented directly on top of P-Trees in PAM, most of them using only 100 lines of code. This versatility and generality simplify algorithm design and programming. First of all, this minimizes the coding effort to re-create the tree structure when necessary (in another programming language, for example). Furthermore, this reduces the incremental effort for users to adapt our tree structure simply as a black box to their own applications.

With all arguments above comes the statement of this thesis.

Thesis Statement . *Using* join *as a primitive yields efficient parallel algorithms and implementations for balanced binary trees, that enables simplicity, theoretical and practical efficiency, balancing-scheme-independency, generic augmentation, persistence, and multi-versioning, which is effective and efficient for a wide range of applications.*

Chapter 2

Preliminaries

2.1 Theoretical Analysis

Parallel Cost Model. Our algorithms are based on nested parallelism with nested fork-join constructs and no other synchronization or communication among parallel tasks.[1] All analysis is using the *binary-forking* model [81], which is based on nested fork-join, but only allows for two branches in a forking. All algorithms are deterministic[2]. To analyze asymptotic costs of a parallel algorithm we use *work* W and *span* S (or *depth* D), where work is the total number of operations and depth is the length of the critical path. In the algorithm descriptions the $s_1 \mid\mid s_2$ notation indicates that statement s_1 and s_2 can run in parallel (and obviously can run in order as in the sequential setting). For theoretical analysis, we use the simple composition rules $W(e_1 \mid\mid e_2) = W(e_1) + W(e_2) + 1$ and $S(e_1 \mid\mid e_2) = \max(S(e_1), S(e_2)) + 1$. 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) [169] with P processors and a greedy scheduler [83, 88, 143]. We assume concurrent reads and exclusive writes (CREW).

Notation. We call a key-value pair in a map an *entry* denoted as $e = (k, v)$. We use $k(e)$ and $v(e)$ to extract the key and the value from an entry. We use $\langle P \rangle$ for a sequence of elements of type P. We use $\langle \cdot, \cdot \rangle$ to denote a pair (similarly for tuples and sequences).

Others. We use *with high probability* (w.h.p.) to mean that for an input of size n the bound holds with probability at least $1 - 1/n^c$ ($c > 0$ is a constant).

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

[2] Note that the bounds and the data structure themselves are not necessarily deterministic. For example, the treaps depends on random priorities. However as long as the priorities are known (independently of the algorithms), the algorithm does not use randomization.

2.2 Balanced Binary Trees

A *binary tree* is either a *nil-node*, or a node consisting of a *left* binary tree T_l, an entry e, and a *right* binary tree T_r, and denoted $node(T_l, e, T_r)$. The entry can be simply a key, or more data associated to it. The *size* of a binary tree, or $|T|$, is 0 for a *nil-node* and $|T_l| + |T_r| + 1$ for a $node(T_l, e, T_r)$. 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 *nil-node*, and $\max(h(T_l), h(T_r)) + 1$ for a $node(T_l, e, T_r)$. *Parent*, *child*, *ancestor* and *descendant* are defined as usual (ancestor and descendant are inclusive of the node itself). We use $lc(T)$ and $rc(T)$ to extract the left and right child (subtree) of T, respectively. A node has *depth* d if taking its parent d times returns the root. Different from some previous work, we use the *nil-node* to refer to an external (empty) node with no data stored in it. A node is called a *leaf* when its both children are *nil nodes*. 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* (also referred to as the symmetric order) of a binary tree is the sequence of values returned by an in-order traversal of the tree. When the context is clear, we use a node u to refer to the subtree T_u rooted at u, and vice versa.

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 we consider four balancing schemes that ensure the height of every tree of size n is bounded by $O(\log n)$. When keys have a total order and the in-order of the tree is consistent with the order, then we call it a *binary search tree (BST)*. We note that the balancing schemes defined below, although typically applied to BSTs, do not require that the binary tree be a search tree.

AVL Trees [12]. AVL trees have the invariant that for every $node(T_l, e, T_r)$, 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.

Red-black (RB) Trees [42]. RB trees 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. All *nil nodes* are always black. Unlike some other presentations, we do not require that the root of a tree is black. Although this does not affect the correctness of our algorithms, our proof of the work bounds requires allowing a red root. We define the *black height* of a node T, denoted $\hat{h}(T)$ 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 $2\log_2(n + 1)$.

Weight-balanced (WB) Trees [217]. WB trees with parameter α (also called BB[α] trees) maintain for every $T = node(T_l, e, T_r)$ the invariant $\alpha \leq \frac{w(T_l)}{w(T)} \leq 1 - \alpha$. We say two weight-balanced trees T_1 and T_2 have *like* weights if $node(T_1, e, T_2)$ is weight balanced. Any α weight-balanced tree of size n has height at most $\log_{\frac{1}{1-\alpha}} n$. For $\frac{2}{11} < \alpha \leq 1 - \frac{1}{\sqrt{2}}$

Notation	Description
$\lvert T\rvert$	The size of tree T
$h(T)$	The height of tree T
$\hat{h}(T)$	The black height of an RB tree T
$rank(T)$	The rank of tree T
$w(T)$	The weight of tree T (i.e, $\lvert T\rvert + 1$)
$p(T)$	The parent of node T
$k(T)$	The key of node T
$lc(T)$	The left child of node T
$rc(T)$	The right child of node T
$\mathcal{A}(T)$	The augmented value of a node T
$\texttt{expose}(T)$	$\langle lc(T), k(T), rc(T)\rangle$

Table 2.1: Summary of notation.

insertion and deletion can be implemented on weight balanced trees using just single and double rotations [82, 217]. We require the same condition for our implementation of *join*, and in particular use $\alpha = 0.29$ in experiments. We also denote $\beta = \frac{1-\alpha}{\alpha}$, which means that either subtree could not have a size of more than β times of the other subtree.

Treaps [246]. Treaps 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).

The notation we use for binary trees is summarized in Table 2.1. Some concepts in Table 2.1 will be introduced later in this thesis.

2.3 Persistent and Functional Data Structures

Persistent Data Structures. A *persistent* data structure [127] is a data structure that preserves previous versions of itself. When being modified, it always creates a new updated structure. For BSTs, persistence can be achieved by path-copying [242]. In path-copying, only the affected path related to the update is copied, so the asymptotical cost for an update remains unchanged.

Purely Functional Data Structures. a purely functional data structure is a data structure that can be implemented in a purely functional language. A purely functional data structure is (strongly) immutable. Obviously all purely functional data structures are persistent. This can be abstracted as the *pure LISP machine* [45, 219, 232] (PLM), which, like the random access machine model (RAM), has some constant number of registers. However, the only instructions for manipulating memory, are (1) a $\texttt{tuple}(v_1, \ldots, v_l)$ instruction, which takes l registers (for some small constant l) and creates a tuple in memory containing their values, and (2) a $\texttt{nth}(t, i)$ instruction, which, given a pointer t

to a tuple and an integer i (both in registers), returns the i-th element in this tuple. Values in the registers and tuples are either primitive, or a pointer to another tuple. There is no instruction for modifying a tuple. This thesis focuses on binary trees, which means that $l = 3$, and each tuple (tree node) maintains three elements: the left child pointer, the right child pointer, and the data in this node. Changing a data structure using PLM instructions are done via *path copying*, meaning that to change a node, its ancestors in the data structure must be copied into new tuples, but the remainder of the data remains untouched. Using PLM instructions, one can create a DAG in memory, which we refer to as the *memory graph*. An example of using path-copying to insert a value into a binary tree memory graph is shown in Figure 1.1(b).

2.4 Computational Geometry

Notation. In two dimensions, let X, Y and $D = X \times Y$ be the types of x- and y-coordinates and the type of points, where X and Y are two sets with total ordering defined by $<_X$ and $<_Y$ respectively. For a point $p \in D$ in two dimensions, we use $x(p) \in X$ and $y(p) \in Y$ to extract its x- and y-coordinates, and use a pair $(x(p), y(p))$ to denote p. For simplicity, we assume all input coordinates are unique. Duplicates can be resolved by slight variations of algorithms in this thesis. We use two endpoints to represent a segment. We use two diagnose points to represent a rectangle.

Sweepline Algorithms. A sweepline algorithm (or plane sweep algorithm) is an algorithmic paradigm that uses a conceptual sweep line to process elements in order [249]. It uses a virtual line sweeping across the plane, which stops at some points (e.g., the endpoints of segments) to make updates. We refer to the points as the *event points*. A more detailed overview and formalization of such sweepline algorithms are in Section 9.2.3.

Problem Definitions. In this thesis, we study several computational geometry problem. In one dimension, we study the stabbing query. For a set S of input intervals on the number line (represented as two endpoints) and a query point q, the stabbing query asks about some information about the subset $S' \in S$ containing the intervals that overlap q. In particular, the query can be about whether S' is empty, the size of S', or all the elements in S'. In two dimensions, this thesis studies the range, segment and rectangle queries. We focus on 2D Euclidean space.

The range query problem is to maintain a set of points, and to answer queries regarding the points contained in a query rectangle.

The segment query problem is to maintain a set of non-intersecting segments, and to answer questions regarding all segments intersected with a query vertical line.

The rectangle stabbing query (also referred to as the *enclosure stabbing query*) problem is to maintain a set of rectangles, and to answer questions regarding rectangles containing a query point.

For all problems, we discuss queries of both listing all queried elements (the *list-all* query), and returning the count of queried elements (the *counting* query). Some other queries, can be implemented by variants (e.g., the weighted sum of all queried elements) or combinations (e.g. rectangle-rectangle intersection queries) of the queries in this thesis.

2.5 Database Management Systems

Transaction Isolation Levels. A transaction's isolation level determines what anomalies it may be exposed to during its execution. These were originally defined in the context of pessimistic two-phase locking concurrency control in the 1990s. *Snapshot isolation* (SI) is an additional level that was proposed later after the original standard was released [59]. SI is a popular isolation level and is often good enough for HTAP environments because its OLAP queries will be read-only. Marking an OLAP transaction as read-only means that the database does not need to maintain its read-write set while it executes. All transactions still check the visibility of each tuple. A DBMS is serializable if its outcome of executing any concurrent transactions (e.g., the resulting state) is equal to executing its transactions in a serial order.

OLTP, OLAP and HTAP Database Systems. Online transaction processing (OLTP) describes systems that facilitate and manage transaction-oriented applications. Such transactions focus on updates that commits changes to the database. OLTP usually requires the system to respond immediately to user requests. Online analytical processing, or OLAP, is an approach to answer multi-dimensional analytical (MDA) queries swiftly in computing. It aims at answering complex analytical and ad hoc queries with a high throughput. Whereas OLTP systems process all kinds of queries (read, insert, update and delete), OLAP is generally optimized for read only and might not even support other kinds of queries. More recently, the term *hybrid transaction/analytical processing (HTAP)* is proposed to "break the wall" between transaction processing and analytics. It means that while transactions makes changes to the database and get timely response, the database can also conduct complex analytical queries correctly and efficiently.

Part I

Methodology

Chapter 3

Join-based Algorithms

In this chapter, we will present the *join*-based algorithms on balanced binary trees. We will abstract out the required rule for a balancing scheme that ensures the correctness and efficiency of *join*-based algorithms. These rules apply to at least four balancing schemes: AVL trees, RB trees, WB trees and treaps.

All *join*-based aggregated functions, such as *build*, *union*, *intersection*, *filter*, etc., can run in parallel. All these algorithms are designed based on the primitive *join*, which captures all balancing criteria such that it is the only function dealing with rebalancing and rotations. Making use of *join*, all the other functions can be expressed in a manner that is generic across balancing schemes, and thus share the same code in implementation. We summarize the common conditions making a balancing scheme *joinable*. This means to associate with each balancing scheme, an efficient *join* algorithm and a effective *rank* function reflecting the balancing criteria, such that they satisfy several rules. As a result, all algorithms and theoretical analysis in this chapter are generic and applicable to all such joinable trees.

Almost all the *join*-based algorithms introduced in this chapter are theoretically-efficient in that they have optimal work. Furthermore, all of them have poly-logarithmic span which implies a great amount of parallelism. Although it is relatively straightforward to get optimal $O(\log n)$ work algorithms for *find*, *insert*, and *delete* using balanced trees, achieving this for the aggregate functions is more challenging. For example, for two trees of sizes m and n where $m \leq n$, the *join*-based set-set operations (*union*, *intersection*, *difference*) on joinable-trees take no more than $O(m \log(n/m + 1))$ work. This is optimal under the comparison model [164]. This work bound is sublinear in the total size of the inputs, and when m is significantly smaller than n this bound is significantly less than $O(m + n)$, and is obviously smaller than $O(m \log n)$. This bound implies that taking unions (or intersections or differences) of two unequal sized ordered sets can be significantly faster than array-based implementations, and is also faster than inserting all elements in the small set (tree) into the larger one. This is especially useful when inserting/deleting a small batch of elements to the current database using union, and the work is only proportional to the size of the batch, but never asymptotically exceed the size of the large batch. These algorithms are also parallel with poly-logarithmic span. For these set functions, although similar algorithms has been introduced in previous research, to the best of our knowledge, there has been no detailed analysis of the efficiency of those algorithms other than [71], which analyzed the work and span only for treaps. Besides their algorithm is different from the *join*-based algorithms in the sense that they always use the heavier root as the pivot. We will show a detailed analysis for the *join*-based set-set algorithms in this section. The analysis is a relatively general proof across different balancing schemes, and is very different from and much simpler than [71].

The function *join* (T_l, e, T_r) takes two binary trees T_l and T_r, and an entry e, and returns a new binary tree for which the symmetric order is a concatenation of the symmetric order of T_l, then e, and then the symmetric order of T_r. If the binary trees are BSTs, then the *join* function connects all entries in the input in order. However, binary trees can also be used to represent sequences, and *join* might be useful for sequences. We use a generalized version of *join* that works for any binary tree, whether a BST or not. We call the middle entry e the *pivot* of the corresponding *join* function. It can be a single key for sequences and sets, or a key-value pair for maps or augmented maps. In the implementation, the middle entry e is usually provided as a pointer to a tree node storing the corresponding key and value.

Generally, beyond *join*, the only access to the tree that the algorithms in this chapter make use of is through *expose*$(\cdot)$, *size*$(\cdot)$ and the comparison function on keys $<_K$. For simplicity, we use $<$, $>$, $\leq$, and $\geq$ as the standard notation to represent the result of comparing two keys based $<_K$. We also use $e(\cdot)$, $k(\cdot)$ and $v(\cdot)$ to extract the entry, the key and the value, respectively. All the update (write) operations are done through *new_node*$(\cdot)$ and *update_value*$(\cdot, \cdot)$. All these functions (*expose*, $<_K$, *new_node* and *update_value*) basically only read the root and we assume they all take constant time. Note that this may not be true for $<_K$ and *new_node* for very complicated key types, but we assume so for the convenience of theoretical analysis.

All the *join*-based algorithms can also be made purely-functional, in which all updates are done through copies. This also makes the tree a persistent [127] data structure. In that case, *update_value*$(\cdot, \cdot)$ is disallowed. We will present more details about persistence in Chapter 5.

We start with presenting the rules for joinable trees with some useful lemmas and theorems. Then we present the *join* algorithms on trees in Section 3.2. We then present the parallel set-set algorithms (*union*, *intersection* and *difference*) along with the proof of its work and span in Section 3.3.2. Some other *join*-based algorithms will be presented in Section 3.3.4. We then discuss how to extend some algorithms to support a combine function on values in Section 3.3.5. Finally we show some further discussions about the extension of the algorithmic framework in Section 3.4.

3.1 Joinable Trees

We here define the preferred properties that makes a tree *joinable*. The concept of joinable trees relies on two subcomponents: a variable for each tree node (subtree) called *rank*, and a proper *join* algorithm. The definition of rank and the *join* algorithm depend on the balancing scheme. The rank of a tree node only relies on the shape of the subtree rooted at it. For a tree T, we denote the rank of it as $rank(T)$.

Definition 1 (Strongly Joinable Trees)**.** *A balancing scheme $\mathcal{S}$ is* strongly joinable*, if we can assign a* rank *for each subtree from $\mathcal{S}$, and there exists a* join (T_1, e, T_2) *on two trees from $\mathcal{S}$, such that the following rules hold:*

1. ***EMPTY RULE**. The rank of a* nil-node *is* 0.
2. ***MONOTONICITY RULE**. For* $C = \text{join}(A, e, B)$*,* $\max(rank(A), rank(B)) \leq rank(C)$.
3. ***SUBMODULARITY RULE**. Suppose* $C = \text{node}(A, e, B)$ *and* $C' = \text{join}(A', e, B')$*. If* $0 \leq rank(A') - rank(A) \leq x$ *and* $0 \leq rank(B') - rank(B) \leq x$*, then* $0 \leq rank(C') - rank(C) \leq x$ *(increasing side). In the other direction, if* $0 \leq rank(A) - rank(A')$ *and* $0 \leq rank(B) - rank(B')$*, then* $0 \leq rank(C) - rank(C')$ *(decreasing side).*
4. ***COST RULE**.* $\text{join}(A, e, B)$ *uses time* $O(|rank(A) - rank(B)|)$.
5. ***BALANCING RULE**. For a node A,*

$$\max(rank(\text{lc}(A)), rank(\text{rc}(A))) + c_l \leq rank(A) \leq \min(rank(\text{lc}(A)), rank(\text{rc}(A))) + c_u$$

where $c_l \leq 1$ and $c_u \geq 1$ are constants.

The last rule about balancing says that the ranks of a child and its parent cannot differ by much. This is not true for some randomization-based balancing schemes such as treaps. In fact, we define a weakly joinable tree as follows.

Definition 2 (Weakly Joinable Trees)**.** *A balancing scheme $\mathcal{S}$ is* weakly joinable*, if it satisfies the **empty rule**, **monotonicity rule** and **submodularity rule** in Definition 2, and the **relaxed balancing rule** and **weak cost rule** as follows:*

- ***RELAXED BALANCING RULE**. There exist constants c_l, c_r, $0 < p_l \leq 1$ and $0 < p_u \leq 1$, such that for a node A and any of its child B:*
  1. *$rank(B) \leq rank(A) - c_l$ happens with probability at least p_l.*
  2. *$rank(B) \geq rank(A) - c_u$ happens with probability at least p_u.*
- ***WEAK COST RULE**.* $\text{join}(A, e, B)$ *uses time* $O(rank(A) + rank(B))$ *w.h.p.*

In this thesis, we use *joinable* to refer to weakly joinable trees. Later in this section, we will show that AVL trees, RB trees and WB trees are strongly joinable, and treaps are weakly joinable.

We also say a tree T is strongly (weakly) joinable if T is from a strongly (weakly) joinable balancing scheme.

Here we present some properties of joinable trees.

Theorem 3.1.1. *For a strongly joinable tree T, $c_l h(T) \leq rank(T) \leq c_u h(T)$.*

This can be inferred directly from the **balancing rule**.

Theorem 3.1.2 (Logarithmic rank for strongly joinable balanced trees). *For a strongly joinable tree T, $rank(T) = O(\log w(T))$, where $w(T) = |T| + 1$ is T's weight.*

Proof. Let $f(r)$ be a function denoting the minimum weight that T can have when T has rank r, i.e., $f(r) = \min_{rank(T)=r} w(T)$. We now look at the two children of T, noted as L and R. WLOG assume $w(L) \geq w(R)$. In the **balancing rule**, we know that $rank(T) - c_u \leq rank(R) \leq rank(T) - c_l$. The total weight of T is at most $2w(R)$. In other words,

$$f(r) \geq 2 \min_{r-c_u \leq r' \leq r-c_l} f(r')$$

This means that $f(r) \geq c \cdot 2^{r/c_u}$ for some constant c. Thus $rank(T)$ is at most $O(\log w(T))$. □

From the above lemma we have:

Theorem 3.1.3 (Logarithmic height for strongly joinable balanced trees). *For a strongly joinable tree T with weight n, $h(T) = O(rank(T)) = O(\log n)$.*

From the **balancing rule** we have:

Theorem 3.1.4 (Similar ranks for balanced strongly joinable trees). *For a strongly joinable tree $T = \mathrm{node}(\mathrm{lc}(T), e, \mathrm{rc}(T))$, then $rank(\mathrm{lc}(T))$ and $rank(\mathrm{rc}(T))$ differ by at most a constant.*

This means that when two strongly joinable trees are balanced with each other, their ranks differ by at most a constant.

For weakly joinable trees, we also have similar property for rank and height.

Theorem 3.1.5 (Logarithmic height for weakly joinable balanced trees). *For a weakly joinable tree T with weight n, $h(T) = O(rank(T)) = O(\log n)$ w.h.p.*

Proof. First of all, note that for a parent node A and a child node B, $h(A) = h(B) + 1$, but $rank(A) \geq rank(B) + c_l$ happen only with a constant probability. Thus $h(T) = O(rank(T))$.

Consider a node T with rank $r = rank(T)$. We follow the path and always go to the smaller subtree (with smaller weight). We call it a *round* every time we go down one level. After m rounds we hit the leaf. Obviously $m \leq \log n$ since every time we go to the smaller side of the subtree.

We note that every such round, the rank increase by no more than c_u with a constant probability. Adding all the m round gets r, which is the rank of T. Based on Chernoff bound, we know that $\Pr[r \geq (1+\delta)c_u p_u m] \leq e^{-\frac{\delta c_u p_u m}{3}}$. Let $(1+\delta)c_u p_u m$ be $c \log n$ for sufficient large c, we have:

$$\begin{aligned}
\Pr[r \geq c \log n] &\leq e^{-\frac{c \log n - c_u m}{3}} \\
&= n^{-c/3} \cdot e^{\frac{c_u p_u m}{3}} \\
&\leq n^{-c + c_u p_u / 3}.
\end{aligned}$$

□

We then extend the **monotonicity rule** and prove the following theorem:

Theorem 3.1.6 (**Bounded rank increasing by *join***). *For a joinable tree* $C = \text{join}(A, e, B)$*, there exists a constant* $c' \leq c_u$*, such that*

$$\max(\mathit{rank}(A), \mathit{rank}(B)) \leq \mathit{rank}(T) \leq \max(\mathit{rank}(A), \mathit{rank}(B)) + c'$$

Proof. The left half of the inequation holds because of the **monotonicity rule**.

For the right half, consider a single tree node $v = \mathit{node}(\mathit{nil\text{-}node}, e, \mathit{nil\text{-}node})$. $\mathit{rank}(v)$ is at most c_u (the **balancing rule**). Consider $v' = \mathit{join}(A, e, B)$, which increases the rank of both side by at most $\max(\mathit{rank}(A), \mathit{rank}(B))$. Then

$$\mathit{rank}(v') \leq \mathit{rank}(v) + \max(\mathit{rank}(A), \mathit{rank}(B)) \leq c_u + \max(\mathit{rank}(A), \mathit{rank}(B))$$

□

We then prove some useful theorems and lemmas that will be used in later proofs of the *join*-based algorithms.

Theorem 3.1.7. *For a joinable tree, suppose a node* T *has rank* r*, then the number of its descendants with rank more than* $r - 1$ *is a constant.*

Proof. From the **balancing rule** we know that after $1/c_l$ generations from T, the rank of the tree node must be smaller than $r - 1$. That is to say, there are at most $2^{1/c_l}$ such nodes in T's subtree with rank no smaller than $r - 1$. □

For any node T, these tree nodes with rank in range $(\mathit{rank}(T) - 1, \mathit{rank}(T)]$ form a contiguous tree-structured component. We call this component the *rank cluster* of T.

Definition 3. *In a joinable tree, we say a node* T *in layer* i *if* $i \leq \mathit{rank}(T) < i + 1$.

Definition 4. *In a joinable tree, we say a node* v *is a rank root, or a rank(*i*)-root if* v *in layer* i *and* v*'s parent is not in layer* i.

We use $s_i(T)$ to denote the number of rank(i)-root nodes in tree T. When the context is clear, we simply use s_i. We use $d(v)$ of a rank root v to denote the size of its rank cluster.

Definition 5. *In a BST, a set of nodes V is called* ancestor-free *if and only if for any two nodes v_1, v_2 in V, v_1 is not the ancestor of v_2.*

Obviously for each layer i, all the rank(i)-root nodes form an ancestor-free set.

The following lemma is important for proving the work bound of the *join*-based set-set algorithms in Section 3.3.3.

Lemma 3.1.8. *There exist a constant t such that for a strongly joinable tree with size N, $s_i \leq \frac{N}{2^{\lfloor i/t \rfloor}}$. More precisely, $t = 1 + \lceil c_u \rceil$.*

Proof. We now organize all rank roots from layer kt to layer $(k+1)t - 1$ into a *group k*, for an integer k. Assume there are s'_k such rank roots in root layer k. We first prove that $s'_k \leq s'_{k-1}/2$, which indicates that $s'_k \leq \frac{N}{2^k}$.

For a node u in group k, we will show that its subtree contains at least two nodes in group $k-1$, one in each subtree. u's rank is at least $kt - 1$ (because of rounding), but at most $(k+1)t - 1$. For the left subtree (the right one is symmetric), we follow one path until we find a node in group k but its children v group smaller than k, so $rank(v) \leq kt - 1$. We will prove that v must be in group $k-1$. Because of the **balancing rule**, $rank(u) - c_u \leq rank(v) \leq kt - 1$. Considering the range of u's rank, we have: $(k-1)t \leq rank(v) \leq kt - 1$. This means that v is in group $k-1$. Therefore, every node in group k corresponds to at least two nodes in group $k+1$. This proves $s'_k \leq s'_{k-1}/2$.

Obviously, each set of rank(i)-root is a subset of the corresponding group. This proves the above lemma. □

Figure 3.11 (b) shows an example of the layers of an AVL tree, which we set the rank of a node to be its height. Because the rank of all AVL nodes are integers, all nodes are rank roots. Theorem 3.1.8 says that from bottom up, every three layers, the number of nodes in an AVL shrinks by a half.[1]

3.2 The *join* Algorithms and Ranks for Each Balancing Scheme

Here we describe algorithms for *join* for the four balancing schemes we defined in Chapter 2, as well as define the rank for each of them. We will then prove they are joinable. For *join*, the pivot can be either just the data entry (such that the algorithm will create a new tree node for it), or a pre-allocated tree node in memory carrying the corresponding data entry (such that the node may be reused, allowing for in-place updates).

[1] Actually, consider that there is no rounding issue for AVL trees, this means that from bottom up, every two layers, the number of nodes in an AVL shrinks by a half.

As mentioned in the introduction and the beginning of this chapter, *join* fully captures what is required to rebalance a tree and can be used as the only function that knows about and maintains the balance invariants. For AVL, RB and WB trees we show that *join* takes work that is proportional to the difference in rank of the two trees. For treaps the work depends on the priority of k. All the *join* algorithms are sequential so the span is equal to the work. We show in this thesis that the *join* algorithms for all balancing schemes we consider lead to optimal work for many functions on maps and sets.

3.2.1 AVL Trees

```
 1  joinRightAVL($T_l, k, T_r$) {
 2    $(l, k', c)$ = expose($T_l$);
 3    if $h(c) \leq h(T_r) + 1$ then {
 4      $T'$ = node($c, k, T_r$);
 5      if $h(T') \leq h(l) + 1$ then return node($l, k', T'$);
 6      else return rotateLeft(node($l, k'$, rotateRight($T'$)));
 7    } else {
 8      $T'$ = joinRightAVL($c, k, T_r$);
 9      $T''$ = node($l, k', T'$);
10      if $h(T') \leq h(l) + 1$ then return $T''$; else return rotateLeft($T''$); }}

11  join($T_l, k, T_r$) {
12    if $h(T_l) > h(T_r) + 1$ then return joinRightAVL($T_l, k, T_r$);
13    else if $h(T_r) > h(T_l) + 1$ then return joinLeftAVL($T_l, k, T_r$);
14    else return node($T_l, k, T_r$); }
```

Figure 3.1: The *join* algorithm on AVL trees – `joinLeftAVL` is symmetric to `joinRightAVL`.

For AVL trees, we define the rank as the height, i.e., $rank(T) = h(T)$. Pseudocode for AVL *join* is given in Figure 3.1 and illustrated in Figure 3.2. Every node stores its own height so that $h(\cdot)$ takes constant time. If the two trees T_l and T_r differ by height at most one, *join* can simply create a new $node(T_l, e, T_r)$. However if they differ by more than one then rebalancing is required. Suppose that $h(T_l) > h(T_r) + 1$ (the other case is symmetric). The idea is to follow the right spine of T_l until a node c for which $h(c) \leq h(T_r) + 1$ is found (line 3). At this point a new $node(c, e, T_r)$ is created to replace c (line 4). Since either $h(c) = h(T_r)$ or $h(c) = h(T_r) + 1$, the new node satisfies the AVL invariant, and its height is one greater than c. The increase in height can increase the height of its ancestors, possibly invalidating the AVL invariant of those nodes. This can be fixed either with a double rotation if the invalid is at the parent (line 6) or a single left rotation if the invalid is higher in the tree (line 10), in both cases restoring the height for any further ancestor nodes. The algorithm will therefore require at most two rotations, as we summarized in the following lemma.

Lemma 3.2.1. *The* join *algorithm in Figure 3.1 on AVL trees requires at most two rotations.*

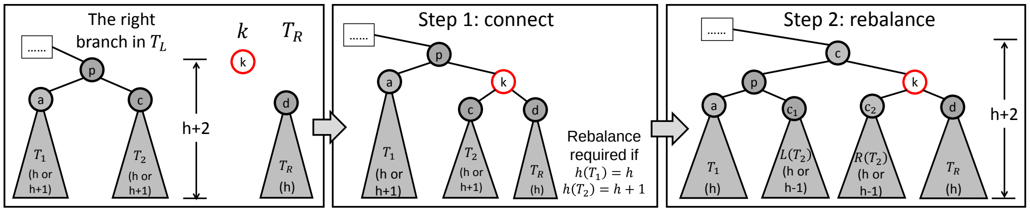


Figure 3.2: An example for *join* on AVL trees – An example for *join* on AVL trees ($h(T_l) > h(T_r) + 1$). We first follow the right spine of T_l until a subtree of height at most $h(T_r) + 1$ is found (i.e., T_2 rooted at c). Then a new $node(c, k, T_r)$ is created, replacing c (Step 1). If $h(T_1) = h$ and $h(T_2) = h + 1$, the node p will no longer satisfy the AVL invariant. A double rotation (Step 2) restores both balance and its original height.

Lemma 3.2.2. *For two AVL trees T_l and T_r, the AVL* join *algorithm works correctly, runs with $O(|h(T_l) - h(T_r)|)$ work, and returns a tree satisfying the AVL invariant with height at most* $1 + \max(h(T_l), h(T_r))$.

Proof. Since the algorithm only visits nodes on the path from the root to c, and only requires at most two rotations (Lemma 3.2.1), it does work proportional to the path length. The path length is no more than the difference in height of the two trees since the height of each consecutive node along the right spine of T_l differs by at least one. Along with the case when $h(T_r) > h(T_l) + 1$, which is symmetric, this gives the stated work bounds. The resulting tree satisfies the AVL invariants since rotations are used to restore the invariant. The height of any node can increase by at most one, so the height of the whole tree can increase by at most one. □

Theorem 3.2.3. *AVL trees are strongly joinable.*

Proof. The AVL invariant and the definition of AVL's rank is ensures the **empty rule** and **balancing rule** ($c_l = 1$, $c_u = 2$). Lemma 3.2.2 ensures the **cost rule** and **monotonicity rule**. The only tricky part is the **submodularity rule**. We prove as follows.

We start with the increasing side.

First note that the ranks of AVL trees are always integers. For $C = node(A, e, B)$ and $C' = join(A', e, B')$, WLOG suppose $rank(A) \geq rank(B)$.

When $0 \leq rank(A') - rank(A) \leq x$ and $0 \leq rank(B') - rank(B) \leq x$, the larger rank of A' and B' is within $rank(A) + x$. Then the rank of C' is in range $[rank(A), rank(A) + x + 1]$ (Lemma 3.2.4). On the other hand, $C = node(A, e, B)$, C is no smaller than $rank(A)+1$ (based on AVL invariants). Thus $0 \leq rank(C) - rank(C') \leq x$ except for when $rank(C') = rank(A)$ and $rank(C) = rank(A) + 1$. We will show that this is impossible.

First of all, $rank(C') = rank(A)$ means that $rank(A') = rank(A)$. Also, $rank(B')$ must be at most $rank(A) - 2$, otherwise the *join* will directly connect A' and B', and $rank(C')$ will be at least $rank(A) + 1$. Considering $rank(B') \geq rank(B)$, this means that A and B cannot be balanced, which leads to a contradiction.

This proves the increasing side of **submodularity rule**. We then prove the decreasing side. Both $rank(A)$ and $rank(B)$ are at most $rank(C) - 1$. Based on Lemma 3.2.2, *join*ing them back gets C' with rank at most $rank(C)$.

□

```
joinRightRB($T_l, k, T_r$) {
  if ($rank(T_l) = \lfloor rank(T_r)/2 \rfloor \times 2$) then return node($T_l, \langle k,$ red$\rangle, T_r$); else {
    ($L', \langle k', c' \rangle, R'$)=expose($T_l$);
    $T'$ = node($L', \langle k', c' \rangle$, joinRightRB($R', k, T_r$));
    if ($c'$=black) and ($color(rc(T')) = color(rc(rc(T')))$=red) then {
      set $rc(rc(T'))$ as black;
      return rotateLeft($T'$);
    } else return $T'$; }}

joinRB($T_l, k, T_r$) {
  if $\lfloor rank(T_l)/2 \rfloor > \lfloor rank(T_r)/2 \rfloor$ then {
    $T'$ =joinRightRB($T_l, k, T_r$);
    if ($color(T')$=red) and ($color(rc(T'))$=red) then return node($lc(T'), \langle k(T'),$ black$\rangle, rc(T')$);
    else return $T'$;
  } else if $\lfloor rank(T_r)/2 \rfloor > \lfloor rank(T_l)/2 \rfloor$ then {
    $T'$ =joinLeftRB($T_l, k, T_r$);
    if ($color(T')$=red) and ($color(lc(T'))$=red) then return node($lc(T'), \langle k(T'),$ black$\rangle, rc(T')$);
    else return $T'$;
  } else {
    if (k is a increase-2 node) then
      return node($T_l, \langle k,$ black$\rangle, T_r$);
    else if ($color(T_l)$=black) and ($color(T_r)$=black)
      return node($T_l, \langle k,$ red$\rangle, T_r$);
    else return node($T_l, \langle k,$ black$\rangle, T_r$); }
}
```

Figure 3.3: The *join* algorithm on red-black trees – The *join* algorithm on red-black trees. joinLeftRB is symmetric to joinRightRB.

3.2.2 Red-black Trees

In RB trees $rank(T) = 2(\hat{h}(T) - 1)$ if T is black and $rank(T) = 2\hat{h}(T) - 1$ if T is red. Tarjan describes how to implement the *join* function for red-black trees [265]. Here we describe a variant that does not assume the roots are black (this is to bound the increase in rank by *union*). The pseudocode is given in Figure 3.3. We store at every node its black height $\hat{h}(\cdot)$. Also, we define the *increase-2* node as a black node, whose both children are also black. This means that the node increases the rank of its children by 2. In the algorithm, the first case is when $\hat{h}(T_r) = \hat{h}(T_l)$. Then if the input node is a increase-2 node, we use it as a black node and directly concatenate the two input trees. This increases the rank of the input by at most 2. Otherwise, if both $root(T_r)$ and $root(T_l)$ are black, we create red $node(T_l, e, T_r)$. When either $root(T_r)$ or $root(T_l)$ is red, we create black $node(T_l, e, T_r)$.

The second case is when $\hat{h}(T_r) < \hat{h}(T_l) = \hat{h}$ (the third case is symmetric). Similarly to AVL trees, *join* follows the right spine of T_l until it finds a black node c for which $\hat{h}(c) = \hat{h}(T_r)$. It then creates a new red $node(c, k, T_r)$ to replace c. Since both c and T_r have the same height, the only invariant that can be violated is the red rule on the root of T_r, the new node, and its parent, which can all be red. In the worst case we may have three red nodes in a row. This is fixed by a single left rotation: if a black node v has $rc(v)$ and $rc(rc(v))$ both red, we turn $rc(rc(v))$ black and perform a single left rotation on v, turning the new node black, and then performing a single left rotation on v. The update is illustrated in Figure 3.4. The rotation, however can again violate the red rule between the root of the rotated tree and its parent, requiring another rotation. Obviously the triple-red issue is resolved after the first rotation. Therefore, expect the bottommost level, a triple-red issue does not happen. The double-red issue might proceed up to the root of T_l. If the original root of T_l is red, the algorithm may end up with a red root with a red child, in which case the root will be turned black, turning T_l rank from $2\hat{h} - 1$ to $2\hat{h}$. If the original root of T_l is black, the algorithm may end up with a red root with two black children, turning the rank of T_l from $2\hat{h} - 2$ to $2\hat{h} - 1$. In both cases the rank of the result tree is at most $1 + rank(T_l)$.

We note that the rank of the output can increase the larger rank of the input trees by 2 only when the pivot is an increase-2 node and the two input trees are balanced both with black roots. This additional condition is to guarantee the **submodularity rule** for RB trees.

Lemma 3.2.4. *For two RB trees T_l and T_r, the RB* join *algorithm works correctly, runs with $O(|rank(T_l) - rank(T_r)|)$ work, and returns a tree satisfying the red-black invariants and with rank at most $2 + \max(rank(T_l), rank(T_r))$.*

Proof. The base case where $h(T_l) = h(T_r)$ is straight-forward. For symmetry, here we only prove the case when $h(T_l) > h(T_r)$. We prove the proposition by induction.

We first show the correctness. As shown in Figure 3.4, after appending T_r to T_l, if p is black, the rebalance has been done, the height of each node stays unchanged. Thus the RB tree is still valid. Otherwise, p is red, p's parent g must be black. By applying a left rotation on p and g, we get a balanced RB tree rooted at p, except the root p is red. If p is the root of the whole tree, we change p's color to black, and the height of the whole tree increases by 1. The RB tree is still valid. Otherwise, if the current parent of p (originally g's parent) is black, the rebalance is done here. Otherwise a similar rebalance is required on p and its current parent. Thus finally we will either find the current node valid (current red node has a black parent), or reach the root, and change the color of root to be black. Thus when we stop, we will always get a valid RB tree.

Since the algorithm only visits nodes on the path from the root to c, and only requires at most a single rotation per node on the path, the overall work for the algorithm is proportional to the depth of c in T_r. This in turn is no more than twice the difference in black height of the two trees since the black height decrements at least every two nodes along the path. This gives the stated work bounds.

For the rank, note that throughout the algorithm, before reaching the root, the black rule is never invalidated (or is fixed immediately), and the only invalidation occurs on the red rule. If the two input trees are originally balanced, the rank increases by at most 2. The only case that the rank increases by 2 is when k is from an increase-2 node, and both $root(T_r)$ and $root(T_l)$ are black.

If the two input tree are not balanced, the black height of the root does not change before the algorithm reaching the root (Step 3 in Figure 3.4). There are then three cases:

1. The rotation does not propagate to the root, and thus the rank of the tree remains as $\max(\hat{h}(T_l), \hat{h}(T_r))$.
2. (Step 3 Case 1) The original root color is red, and thus a double-red issue occurs at the root and its right child. In this case the root is colored black. The black height of the tree increases by 1, but since the original root is red, the rank increases by only 1.
3. (Step 3 Case 1) The original root color is black, but the double-red issue occurs at the root's child and grandchild. In this case another rotation is applied as shown in Figure 3.4. The black height remains, but the root changed from black to red, increasing the rank by 1.

□

Theorem 3.2.5. *RB trees are joinable.*

Proof. The RB invariant and the definition of RB's rank is ensures the **empty rule** and **balancing rule** ($c_l = 1$, $c_u = 2$). Theorem 3.2.4 ensures the **cost rule** and **monotonicity rule**. The only tricky part is the **submodularity rule**. We prove as follows.

We start with the increasing side.

First note that the ranks of RB trees are always integers. For $C = node(A, e, B)$ and $C' = join(A', e, B')$. WLOG suppose $rank(A) \geq rank(B)$.

When $0 \leq rank(A') - rank(A) \leq x$ and $0 \leq rank(B') - rank(B) \leq x$, the larger rank of A' and B' is within $rank(A) + x$. Then the rank of C' is in range $[rank(A), rank(A) + x + 2]$ (Lemma 3.2.4). On the other hand, $C = node(A, e, B)$, C is either $rank(A) + 1$ or $rank(A) + 2$ (because all ranks are integers). Thus $0 \leq rank(C') - rank(C) \leq x$ except for the following cases.

1. $rank(C') = rank(A) = r$.
2. $rank(C') = r + 1$, but $rank(C) = r + 2$.
3. $rank(C') = r + x + 2$, but $rank(C) = r + 1$.

We first show that case 1 is impossible.

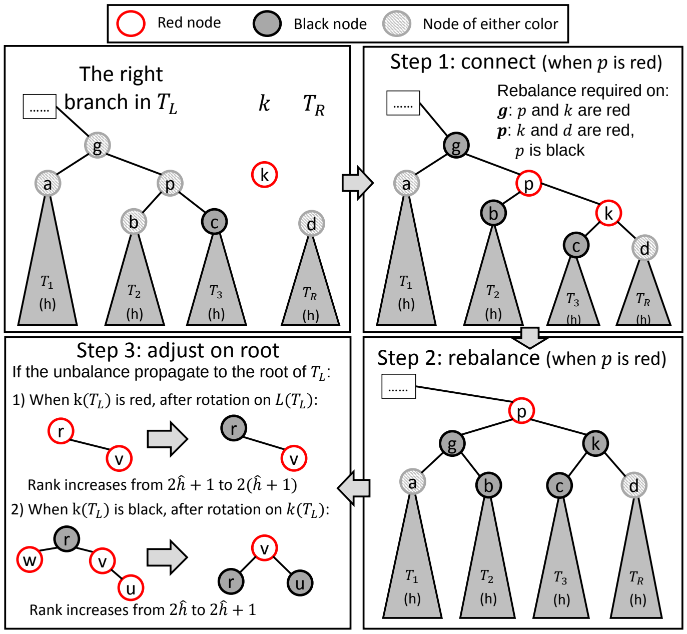

Red node
Black node
Node of either color
The right branch in T_L
k
T_R
g
a
p
b
c
k
d
T_1 (h)
T_2 (h)
T_3 (h)
T_R (h)
Step 1: connect (when p is red)
Rebalance required on:
g: p and k are red
p: k and d are red, p is black
Step 2: rebalance (when p is red)
Step 3: adjust on root
If the unbalance propagate to the root of T_L:
1) When k(T_L) is red, after rotation on $L(T_L)$:
r
v
Rank increases from $2\hat{h}+1$ to $2(\hat{h}+1)$
2) When k(T_L) is black, after rotation on $k(T_L)$:
w
u
Rank increases from $2\hat{h}$ to $2\hat{h}+1$

Figure 3.4: An example of *join* on red-black trees – An example of *join* on red-black trees ($\hat{h} = \hat{h}(T_l) > \hat{h}(T_r)$). We follow the right spine of T_l until we find a black node with the same black height as T_r (i.e., c). Then a new red $node(c, k, T_r)$ is created, replacing c (Step 1). The only invariant that can be violated is when either c's previous parent p or T_r's root d is red. If so, a left rotation is performed at some black node. Step 2 shows the rebalance when p is red. The black height of the rotated subtree (now rooted at p) is the same as before ($h + 1$), but the parent of p might be red, requiring another rotation. If the red-rule violation propagates to the root, the root is either colored red, or rotated left (Step 3).

First of all, $rank(A') \geq rank(A) = rank(C')$. On the other hand $rank(C') \geq rank(A')$ because $C' = join(A', e, B')$. Therefore $rank(A') = rank(C') = rank(A)$.

Also, $rank(B)$ is at least $r - 1$ because A and B are balanced. Considering $rank(B') \geq rank(B)$, $rank(B')$ is at least $r - 1$, but at most $rank(C') = r$.

If A' and B' are balanced, the rank of C' is at least $rank(A) + 1$, which leads to a contradiction.

If A' and B' are not balanced, but their ranks differ by only 1. This is only possible when one of them has black height h with a black root, and the other one has black height $h - 1$ with a red root. However, the *join* algorithm result in double-red on the root's child and its grandchild. After fixing it the rank also increases by 1, which leads to a contradiction.

We then show that case 2 is impossible.

First of all, $rank(B)$ is at least $rank(A) - 1$ because A and B are balanced. Considering $rank(B') \geq rank(B)$, $rank(B')$ is at least $rank(A') - 1$.

If $rank(C) = r + 2$, C must be an increase-2 node. This means that $rank(A) = rank(B) = r$, and they are both black. $rank(A') \geq rank(A) = rank(C') - 1$. On the other hand $rank(C') \geq rank(A')$ because $C' = join(A', e, B')$. Therefore $rank(A') = rank(C')$ or $rank(A') = rank(C') - 1 = rank(A)$.

1. If $rank(A') = rank(C')$, from the same statement of case 1, we can show this is impossible. This is because the *join* algorithm will always lead to the result where C' has rank larger than A'.
2. If $rank(A') = rank(C') - 1 = rank(A)$, A' must be black since $rank(A)$ is even. In this case, $rank(B')$ is r, or $r + 1$.

   If $rank(A') = rank(B') = r$, B' is also black. In this case the algorithm will result in $C' = r + 2$. This leads to a contradiction.

   If $rank(B') = r + 1$, B' is red but is still balanced with A'. In this case, the algorithm also results in C' with rank $r + 2$. This also leads to a contradiction.

Finally we prove that case 3 is impossible.

$rank(C') = r + x + 2$ means that A', B' and C' are all black. Also A' and B' must both have rank $r + x$ and black roots. This means that C' (and also C) is an increase-2 node. Thus $rank(C)$ must be $r + 2$, which leads to a contradiction.

This proves the increasing side of the **submodularity rule**. Next we look at the decreasing side. There are two cases.

1. Both $rank(A') < rank(A)$ and $rank(B') < rank(B)$ hold. They have to decrease by at least one because the rank of a RB tree is always integer. We know that $rank(C') \leq \max(rank(A'), rank(B')) + 2$.

   If $rank(C') = \max(rank(A'), rank(B')) + 2$, C' must be a increasing-2 node. $rank(C)$ is $rank(A) + 2$ (also $rank(B) + 2$). Also, A, B, C, A', B' and C' are all black. Then $rank(A')$ and $rank(B')$ can be at most $rank(A) - 2$ and $rank(B) - 2$, respectively. *join*ing A' and B' increase the maximum rank by at most 2. Therefore $rank(C')$ is no more than $rank(C)$ holds.

   If $rank(C') \leq \max(rank(A'), rank(B')) + 1$, *join*ing them back results in an output tree of rank at most $\max(rank(A'), rank(B')) + 1$. This proves $rank(C') \leq rank(C)$.
2. Either $rank(A') = rank(A)$ or $rank(B') = rank(B)$. WLOG assume $rank(A') = rank(A)$ and $rank(B') \leq rank(B)$. There are three cases.
   (a) A (so is A') is black with rank $2h$, and C is red with rank $2h + 1$. $rank(B') \leq rank(B) = 2h$. Therefore $rank(C') \leq rank(A') + 1 = rank(C)$.
   (b) A (so is A') is black with rank $2h$, and C is black with rank $2h + 2$. $rank(B') \leq rank(B) \leq 2h + 1$. When $rank(B') \leq 2h$, $rank(C') \leq rank(A') + 2 = rank(C)$. When $rank(B') = 2h + 1$, B is red and C' is not an increasing-2 node. Therefore we also get $rank(C') \leq rank(B') + 1 = rank(C)$.

(c) A (so is A') is red with rank $2h+1$, and C is black with rank $2h+2$. $rank(B') \leq rank(B) \leq 2h+1$. Therefore $rank(C') \leq rank(A')+1 = rank(C)$.

In summary, the **submodularity rule** holds for RB trees. RB trees are strongly joinable. □

```
joinRightWB($T_l, k, T_r$) {
  ($l, k', c$)=expose($T_l$);
  if (balance($|T_l|, |T_r|$) then return node($T_l, k, T_r$)); else {
    $T'$ = joinRightWB($c, k, T_r$);
    ($l_1, k_1, r_1$) = expose($T'$);
    if like($|l|, |T'|$) then return node($l, k', T'$);
    else if (like($|l|, |l_1|$)) and (like($|l| + |l_1|, r_1$)) then return rotateLeft(node($l, k', T'$));
    else return rotateLeft(node($l, k'$,rotateRight($T'$))); }}

joinWB($T_l, k, T_r$) {
  if heavy($T_l, T_r$) then return joinRightWB($T_l, k, T_r$);
  else if heavy($T_r, T_l$) then return joinLeftWB($T_l, k, T_r$);
  else return node($T_l, k, T_r$); }
```

Figure 3.5: The *join* algorithm on weight-balanced trees – `joinLeftWB` is symmetric to `joinRightWB`.

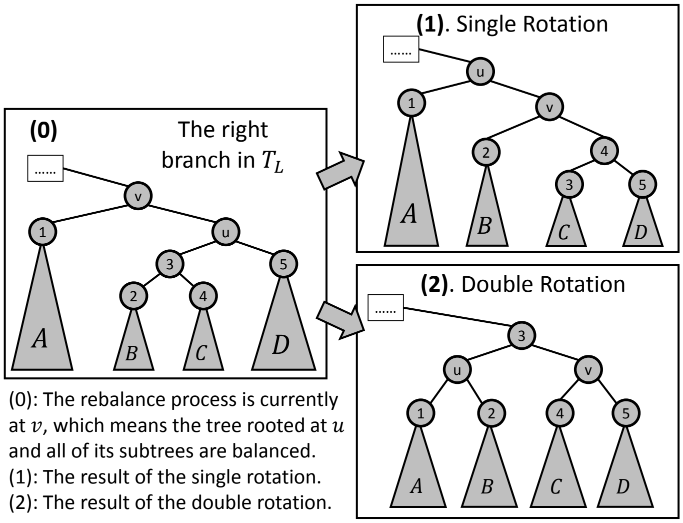
3.2.3 Weight Balanced Trees

For WB trees $rank(T) = \log_2(w(T)) - 1$. We store the weight of each subtree at every node. The algorithm for joining two weight-balanced trees is similar to that of AVL trees and RB trees. The pseudocode is shown in Figure 3.5. The *like* function in the code returns true if the two input tree sizes are balanced based on the factor of α, and false otherwise. If T_l and T_r have like weights the algorithm returns a new $node(T_l, e, T_r)$. Suppose $|T_r| \leq |T_l|$, the algorithm follows the right branch of T_l until it reaches a node c with like weight to T_r. It then creates a new $node(c, r, T_r)$ replacing c. The new node will have weight greater than c and therefore could imbalance the weight of c's ancestors. This can be fixed with a single or double rotation (as shown in Figure 3.6) at each node assuming α is within the bounds given in Chapter 2.

Lemma 3.2.6. *For two α weight-balanced trees T_l and T_r and $\alpha \leq 1 - \frac{1}{\sqrt{2}} \approx 0.29$, the weight-balanced* join *algorithm works correctly, runs with $O(|\log(w(T_l)/w(T_r))|)$ work, and returns a tree satisfying the α weight-balance invariant and with rank at most $1 + \max(rank(T_l), rank(T_r))$.*

The proof is shown in the Appendix.

Notice that this upper bound is the same as the restriction on α to yield a valid weighted-balanced tree when inserting a single node.Then we can induce that when the rebalance process reaches the root, the new weight-balanced tree is valid. The proof is



(0): The rebalance process is currently at v, which means the tree rooted at u and all of its subtrees are balanced.
(1): The result of the single rotation.
(2): The result of the double rotation.

Figure 3.6: An illustration of single and double rotations possibly needed to rebalance weight-balanced trees – In the figure the subtree rooted at u has become heavier due to joining in T_l and its parent v now violates the balance invariant.

intuitively similar as the proof stated in [82, 217], which proved that when $\frac{2}{11} \leq \alpha \leq 1-\frac{1}{\sqrt{2}}$, the rotation will rebalance the tree after one single insertion. In fact, in our *join* algorithm, the "inserted" subtree must be along the left or right spine, which actually makes the analysis easier.

```
joinTreap($T_l, k, T_r$) {
  **if** prior($k, k_1$) **and** prior($k, k_2$) **then return** node($T_l, k, T_r$) **else** {
    ($l_1, k_1, r_1$)=expose($T_l$);
    ($l_2, k_2, r_2$)=expose($T_r$);
    **if** prior($k_1, k_2$) **then return** node($l_1, k_1$,joinTreap($r_1, k, T_r$));
    **else return** node(joinTreap($T_l, k, l_2$),$k_2, r_2$); }}
```

Figure 3.7: The *join* algorithm on treaps – prior(k_1, k_2) decides if the node k_1 has a higher priority than k_2.

Theorem 3.2.7. *WB trees are strongly joinable when $\alpha \leq 1 - \frac{1}{\sqrt{2}} \approx 0.29$.*

Proof. The WB invariant and the definition of WB's rank is ensures the **empty rule** and **balancing rule** ($c_l = \log_{(1-\alpha)} 2$, $c_u = \log_\alpha 2$). Theorem 3.2.6 ensures the **cost rule** and the **monotonicity rule**.

For **submodularity rule**, note that $rank(C) = \log_2(w(A) + w(B))$, and $rank(C') = \log_2(w(A') + w(B'))$. When either of A and B change their weight by more than $\log_2 x$, obviously the total weight of C will not increase or decrease by a factor of $\log_2 x$. □

3.2.4 Treaps

For treaps $rank(T) = \log_2(w(T)) - 1$. The treap *join* algorithm (as in Figure 3.7) first picks the key with the highest priority among k, $k(T_l)$ and $k(T_r)$ as the root. If k is the root then the we can return $node(T_l, k, T_r)$. Otherwise, WLOG, assume $k(T_l)$ has a higher priority. In this case $k(T_l)$ will be the root of the result, $lc(T_l)$ will be the left tree, and $rc(T_l)$, k and T_r will form the right tree. Thus *join* recursively calls itself on $rc(T_l)$, k and T_r and uses result as $k(T_l)$'s right child. When $k(T_r)$ has a higher priority the case is symmetric. The cost of *join* is therefore the depth of the key k in the resulting tree (each recursive call pushes it down one level). In treaps the shape of the result tree, and hence the depth of k, depend only on the keys and priorities and not the history. Specifically, if a key has the t^{th} highest priority among the keys, then its expected depth in a treap is $O(\log t)$ (also w.h.p.). If it is the highest priority, for example, then it remains at the root.

Lemma 3.2.8. *For two treaps T_l and T_r, if the priority of k is the t-th highest among all keys in $T_l \cup \{k\} \cup T_r$, the treap* join *algorithm works correctly, runs with $O(\log t + 1)$ work in expectation and w.h.p., and returns a tree satisfying the treap invariant with rank at most $1 + \max(rank(T_l), rank(T_r))$.*

Theorem 3.2.9. *Treaps are weakly joinable.*

Proof. The **empty rule** and **monotonicity rule** hold from the definition of rank. The **submodularity rule** holds for the same reason as the WB tree.

For the **relaxed balancing rule**, note that the weight of the tree shrink by a factor of 1/3 to 2/3 with probability 1/3. This means that going down from a parent to a child, the rank of a node decrease by a constant between $\log_2 3$ and $\log_2 3/2$ with probability 1/3. This proves the **relaxed balancing rule**.

For **weak cost rule** note that the cost of *join* is at most the height of the larger input tree, which is $O(rank(T))$ w.h.p. □

We also present Lemma 3.2.12, which is useful in the proofs in later section. For a treap node v we use $d(v)$ to denote the number of generations to take from v to reach a node u with rank at most $rank(v) - 1$. The lemma proves that $\mathbb{E}[d(v)]$ is a constant.

Lemma 3.2.10. *If v is a rank root in a treap, $d(v)$ is less than a constant in expectation.*

Proof. Consider a rank(i)-root v that has weight $N \in [2^k, 2^{k+1})$. The probability that $d(v) \geq 2$ is equal to the probability that one of its grandchildren has weight at least 2^k. This probability P is:

$$
\begin{aligned}
P &= \frac{1}{2^k} \sum_{i=2^k+1}^{N} \frac{i-2^k}{i} && (3.1)\\
&\le \frac{1}{2^k} \sum_{i=2^k+1}^{2^{k+1}} \frac{i-2^k}{i} && (3.2)\\
&\approx 1 - \ln 2 && (3.3)
\end{aligned}
$$

We denote $1 - \ln 2$ as p_c. Similarly, the probability that $d(v) \geq 4$ should be less than p_c^2, and the probability shrink geometrically as $d(v)$ increase. Thus the expected value of $d(v)$ is a constant. □

We note that in treaps, each rank cluster is a chain. This is true because if two children of one node v are both in layer i, the weight of v is more than 2^{i+1}, meaning that v should be layer $i+1$. The above lemma means that the expected size of a rank cluster is also a constant.

As a summary of this section, we present the following theorem.

Theorem 3.2.11. *AVL, RB and WB trees are strongly joinable, and treaps are weakly joinable.*

For a treap node v we use $d(v)$ to denote the number of generations to take from v to reach a node u with rank at most $rank(v) - 1$. The lemma proves that $\mathbb{E}[d(v)]$ is a constant.

Lemma 3.2.12. *If v is a rank root in a treap, $d(v)$ is less than a constant in expectation.*

Proof. Consider a rank(i)-root v that has weight $N \in [2^k, 2^{k+1})$. The probability that $d(v) \geq 2$ is equal to the probability that one of its grandchildren has weight at least 2^k. This probability P is:

$$
\begin{aligned}
P &= \frac{1}{2^k} \sum_{i=2^k+1}^{N} \frac{i-2^k}{i} && (3.4)\\
&\le \frac{1}{2^k} \sum_{i=2^k+1}^{2^{k+1}} \frac{i-2^k}{i} && (3.5)\\
&\approx 1 - \ln 2 && (3.6)
\end{aligned}
$$

We denote $1 - \ln 2$ as p_c. Similarly, the probability that $d(v) \geq 4$ should be less than p_c^2, and the probability shrink geometrically as $d(v)$ increase. Thus the expected value of $d(v)$ is a constant. □

We note that in treaps, each rank cluster is a chain. This is true because if two children of one node v are both in layer i, the weight of v is more than 2^{i+1}, meaning that v should be layer $i + 1$. The above lemma means that the expected size of a rank cluster is also a constant.

Function	***Work***	***Span***
insert, *delete*, *update*, *find*, *first*, *last*, *range*, *split*, *join2*, *previous*, *next*, *rank*, *select*, *up_to*, *down_to*	$O(\log n)$	$O(\log n)$
union, *intersection*, *difference*	$O\left(m \log\left(\frac{n}{m} + 1\right)\right)$	$O(\log n \log m)$
map, *reduce*, *map_reduce*, *to_array*	$O(n)$	$O(\log n)$
build, *filter*	$O(n)$	$O(\log^2 n)$

Table 3.1: The core *join*-based algorithms and their asymptotic costs – The cost is given under the assumption that all parameter functions take constant time to return. For functions with two input trees (*union*, *intersection* and *difference*), *n* is the size of the larger input, and *m* of the smaller.

3.3 Algorithms Using *join*

Split

```
split($T, k$) {
  if $T = \emptyset$ then
    return ($\emptyset$,false,$\emptyset$);
  ($L, m, R$) = expose($T$);
  if $k = m$ then return ($L$,true,$R$);
  if $k < m$ then {
    ($T_L, b, T_R$) = split($L, k$);
    return ($T_L$,$b$,join($T_R, m, R$)); }
  ($T_L, b, T_R$) = split($R, k$);
  return (join($L, m, T_L$), $b$, $T_R$); } }
```

join2

```
split_last($T$) { // split_first is symmetric
  ($L, k, R$) = expose($T$);
  if $R = \emptyset$ then return($L, k$);
  ($T', k'$) = split_last($R$);
  return (join($L, k, T'$),$k'$); }
join2($T_l$,$T_r$) {
  if $T_l = \emptyset$ then return $T_r$;
  ($T', k$) = split_last($T_l$);
  return join($T', k, T_r$);
}
```

Figure 3.8: *split* and *join2* algorithms – They are both independent of balancing schemes.

The *join* function, as a subroutine, has been used and studied by many researchers and programmers to implement more general set operations. In this section, we describe algorithms for various functions that use just *join*. The algorithms are generic across balancing schemes. The pseudocodes for the algorithms in this section is shown in Figure 3.10. Beyond *join* the only access to the trees we make use of is through *expose* and *rank*(·), which only read the root. main set operations, which are *union*, *intersection* and *difference*, are optimal (or known as *efficient*) in work. The pseudocode for all the algorithms introduced in this section is presented in Figure 3.12.

3.3.1 Two Helper Functions: *split* and *join2*

We start with presenting two helper functions *split* and *join2*. 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 $k \in T$. *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.

Although both sequential, these two functions, along with the *join* function, are essential for help other algorithms to achieve good parallelism. Intuitively, when processing a tree in parallel, we recurse on two sub-components of the tree in parallel by *split*ing the tree by some key. In many cases, the splitting key is just the root, which means directly using the two subtrees of natural binary tree structure. After the recursions return, we combine the result of the left and right part, with or without the middle key, using *join* or *join2*. Because of the balance of the tree, this framework usually gives high parallelism with shallow span (e.g., poly-logarithmic).

Split. As mentioned above, *split*(T, k) splits a tree T by a key k into T_l and T_r, along with a bit b indicating if $k \in T$. Intuitively, the *split* 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. Writing the code in a recursive manner, this algorithm first determine if k falls in the left (right) subtree, or is exactly the root. If it is the root, then the algorithm directly returns the left and the right subtrees as the two return trees and *true* as the bit b. Otherwise, WLOG, suppose k falls in the left subtree. The algorithm further *split* the left subtree into T_L and T_R with the return bit b'. Then the return bit $b = b'$, the T_l in the final result will be T_L, and T_r means to *join* T_R with the original right subtree by the original root. Figure 3.9 gives an example.

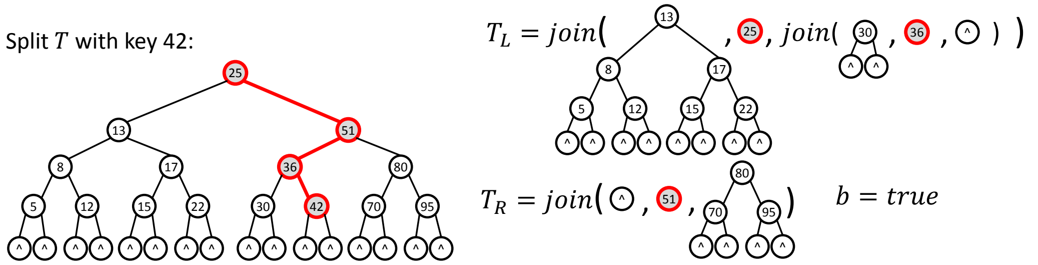


Figure 3.9: An example of *split* in a BST with key 42 – We first search for 42 in the tree and split the tree by the searching path, then use *join* to combine trees on the left and on the right respectively, bottom-top.

The cost of the algorithm is proportional to the rank of the tree, as we summarize and prove in the following theorem.

Theorem 3.3.1. *The work of* split(T, k) *is* $O(h(T))$ *for all strongly joinable trees and treaps. The two resulting trees* T_l *and* T_r *will have rank at most* $rank(T) + c_0$ *for some constant* c_0.

Proof. We only consider the work of joining all subtrees on the left side. The other side is symmetric. Suppose we have l subtrees on the left side, denoted from bottom to top as $T_1, T_2, \ldots T_l$. As stated above, we consecutively join T_1 and T_2 returning T'_2, then join T'_2 with T_3 returning T'_3 and so forth, until all trees are merged. The overall work of *split* is the sum of the cost of $l - 1$ *join* functions. One observation is that, the pivot of the *join* of T'_{i-1} and T_i, denoted as e_i, used to be T_i's parent. Meanwhile, T'_{i-1} is a subset of T_i's original sibling, denoted as X_i. We use $T(e_i) = node(X_i, e_i, T_i)$ to denote the original subtree rooted at e_i in the input.

We first prove by induction that computing $T'_i = join(T'_{i-1}, e_i, T_i)$ gets T'_i with rank no more than $rank(X_{i+1})$.

From the induction hypothesis, we know that $rank(T'_{i-1}) \leq r(X_i)$. Considering $T'_i = join(T'_{i-1}, e_i, T_i)$ and $T(e_i) = node(X_i, e_i, T_i)$, from the **submodularity rule**, we can get $rank(T'_i) \leq rank(T(e_i))$. Considering $T(e_i)$ is a subtree in X_{i+1}, we have

$$rank(T'_i) \leq rank(T(e_i)) \leq rank(X_{i+1}$$

We next prove the cost. The cost of the i-th *join* is $W_i \leq c|rank(T_i) - rank(T'_{i-1})|$. Note that $rank(T'_{i-1}) \leq rank(X_i) \leq rank(T_i) + c_u - c_l$. Also, note that T'_i is achieved by joining T_i and another tree. Therefore, $rank(T_i) - rank(T'_{i-1}) \leq rank(T'_i) - rank(T'_{i-1})$.

This means that either $W_i = c(rank(T'_i) - rank(T'_{i-1}))$, or W_i is a constant no more than $c_2 = c \cdot (c_u - c_l)$. Therefore $W_i \leq c(rank(T'_i) - rank(T'_{i-1}) + 2c_2)$.

$$\begin{aligned}\sum_{i=1}^{h(T)} W_i \leq & \sum_{i=1}^{h(T)} c(rank(T'_i) - rank(T'_{i-1}) + 2c_2)\\ \leq & 2c \cdot c_2 \cdot h(T) + rank(T'_i)\\ \leq & O(h(T)) + rank(T(e_{h(T)})) \leq O(h(T))\end{aligned}$$

For treaps, each *join* uses the key with the highest priority since the key is always on a upper level. Hence by Lemma 3.2.8, the complexity of each *join* is $O(1)$ and the work of split is at most $O(h(T))$. Obviously for treaps we have $rank(T_l)$ and $rank(T_r)$ at most $rank(T)$. □

Join2. As stated above, the *join2* function is defined similar to *join* without the middle entry. The *join2* algorithm first choose one of the the input trees, and extract its last (if it

Union

```
union($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then return $T_2$;
  if $T_2 = \emptyset$ then return $T_1$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = union($L_1$,$L_2$) ||
        $T_r$ = union($R_1$,$R_2$);
  return join($T_l$,$k_2$,$T_r$); }
```

Intersection

```
intersect($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then return $\emptyset$;
  if $T_2 = \emptyset$ then return $\emptyset$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = intersect($L_1$,$L_2$) ||
    $T_r$ = intersect($R_1$,$R_2$);
  if $b$ then return join($T_l$,$k_2$,$T_r$);
  else return join2($T_l$,$T_r$); }
```

Difference

```
difference($T_1$,$T_2$) {
  if $T_1 = \emptyset$ then $\emptyset$;
  if $T_2 = \emptyset$ then $T_1$;
  ($L_2$,$k_2$,$R_2$) = expose($T_2$);
  ($L_1$,$b$,$R_1$) = split($T_1$,$k_2$);
  $T_l$ = difference($L_1$,$L_2$) ||
    $T_r$ = difference($R_1$,$R_2$);
  return join2($T_l$,$T_r$); }
```

Figure 3.10: *join*-based algorithms for set-set operations – They are all independent of balancing schemes. The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

is T_l) or first (if it is T_r) element k. The two cases take the same asymptotical cost. The extracting process is similar to the *split* algorithm. The algorithm then uses k as the pivot to *join* the two trees. In the code shown in Figure 3.8, the *split_last* algorithm first finds the last element k (by following the right spine) in T_l and on the way back to root, *join*s the subtrees along the path. We denote the result of dropping k in T_L as T'. Then *join* (T', 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 3.3.2. *The work of* T =join2(T_l, T_r) *is* $O(r(T_l) + r(T_r))$ *for all joinable trees.* $rank(T) \leq \max(rank(T_l), rank(T_r)) + c'$.

Proof. The cost bound holds because *split_last* and *join* both take work asymptotically no more than the larger tree rank. We next prove the range of $rank(T)$. First of all, splitting the last from T_l only decreases its rank (Theorem 3.3.1). Therefore T =*join* (T', k, T_r) has rank no more than $\max(rank(T_r), rank(T_l)) + c'$. This proves the inequation. □

3.3.2 Set-set Functions Using *join*

In this section, we will present the *join*-based algorithm on set-set functions, including *union*, *intersection* and *difference*. Many other set-set operations, such as symmetric difference, can be implemented by a combination of *union*, *intersection* and *difference* with no extra asymptotical work. We will start with presenting some background of these algorithms, and then explain in details about the *join*-based algorithms. Finally, we show the proof of their cost bound.

Background. The parallel set-set functions are particularly useful when using parallel machines since they can support parallel bulk updates. As mentioned, although supporting efficient algorithms for basic operations on trees, such as insertion and deletion, are rather straightforward, implementing efficient bulk operations is more challenging, especially considering parallelism and different balancing schemes. For example, combining two ordered sets of size n and $m \leq n$ in the format of two arrays would take work $O(m + n)$ using the standard merging algorithm in the merge sort algorithm. This makes even inserting an single element into a set of size n to have linear cost. This is because even most of the chunks of data in the input remain consecutive, the algorithm still need to scan and copy them to the output array. Another simple implementation is to store both sets as balanced trees, and insert the elements in the smaller tree into the larger one, costing $O(m \log n)$ work. It overcomes the issue of redundant scanning and copying, because many subtrees in the larger tree remain untouched. However, this results in $O(n \log n)$ time, for combining two ordered sets of the same size, while it is easy to make it $O(n)$ by arrays. The problem lies in that the algorithm fails to make use of the ordering in the smaller tree.

The lower bound for comparison-based algorithms for *union*, *intersection* and *difference* for inputs of size n and $m \leq n$, and returning an ordered structure[2], is $\log_2 \binom{m+n}{n} = \Theta\big(m \log\big(\frac{n}{m} + 1\big)\big)$ ($\binom{m+n}{n}$ is the number of possible ways n keys can be interleaved with $m$keys). Brown and Tarjan first matched these bounds, asymptotically, using a sequential algorithm based on red-black trees [91]. Although designed for merging, the algorithm can be adapted for *union*, *intersection* and *difference* with the same bounds. The bound is interesting since it shows that implementing insertion with union, or deletion with difference, is asymptotically efficient ($O(\log n)$ time), as is taking the union of two equal sized sets ($O(n)$ time). However, the Brown and Tarjan algorithm is complicated, only works on red-black trees, and completely sequential.

Adams later described very elegant algorithms for union, intersection, and difference, as well as other functions based on *join* [10, 11]. Adams' algorithms were proposed in an international competition for the Standard ML community, which is about implementations on "set of integers". Prizes were awarded in two categories: fastest algorithm, and most elegant yet still efficient program. Adams won the elegance award, while his algorithm is almost as fast as the fastest program for very large sets, and was faster for smaller sets. Because of the elegance of the algorithm, at least three libraries use Adams' algorithms for their implementation of ordered sets and tables (Haskell [200] and MIT/GNU Scheme, and SML). The idea of Adams' algorithm enlightens our parallel *join*-based set-set algorithms and the implementation in the PAM library, for which the sequential version on weight balanced trees is exactly the same as Adams' algorithm.

Although only considered weight-balanced trees, Adams' algorithms actually show that in principle all balance criteria for search trees can be captured by the single function *join*. As long as a valid *join* algorithm on a certain balancing scheme is provided, the correctness of the *join*-based set-set operations can be guaranteed on the corresponding balancing scheme.

[2] By "ordered structure" we mean any data structure that can output elements in sorted order without any further comparisons—e.g., a sorted array, or a binary search tree.

Surprisingly, however, there have been almost no results on bounding the work (time) of Adams' algorithms, in general nor on specific tree types. Adams informally argues that his algorithms take $O(n + m)$ work for weight-balanced tree, but that is a very loose bound. Blelloch and Reid-Miller later show that similar algorithms for treaps [71], are optimal for work (i.e. $\Theta\big(m \log\big(\frac{n}{m} + 1\big)\big)$), and are also parallel. Their algorithms, however, are specific for treaps. The problem with bounding the work of Adams' algorithms, is that just bounding the time of *split*, *join* and *join2* with logarithmic costs is not sufficient.[3] One needs additional properties of the trees. As a result, there is no tight bound even for Adams' original algorithms on weight-balanced trees, not to mention other balancing schemes.

Our work gives the first work-optimal bounds for the *join*-based algorithms for the four balancing schemes considered by this thesis. We show that with appropriate (and simple) implementations of *join* for each balancing scheme, we achieve asymptotically optimal bounds on work. These bounds hold when either input tree is larger (this was surprising to us). Furthermore the algorithms have $O(\log n \log m)$ span, and hence are highly parallel. To prove the bounds on work we show that our implementations of *join* satisfy certain conditions based on a rank we define for each tree type. In particular the cost of *join* must be proportional to the difference in ranks of two trees, and the rank of the result of a join must be at most one more than the maximum rank of the two arguments.

Algorithms. *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). ger) than $k(T_1)$. Then two recursive calls to *union* are made in parallel. One unions $lc(T_2)$ with L_1, returning T_l, and the other one unions $rc(T_2)$ 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 *intersection* (T_1, T_2) and *difference* (T_1, T_2) take the intersection and difference of the keys in their sets, respectively. The algorithms are similar to *union* in that they use one tree to split the other. However, the method for joining and the base cases are different. For *intersection*, *join2* is used instead of *join* if the root of the first *is not* found in the second. Accordingly, the base case for the *intersection* algorithm is to return an empty set when either set is empty. For *difference*, *join2* is used anyway because $k(T_2)$ should be excluded in the result tree. The base cases are also different in the obvious way.

The cost of the algorithms described above can be summarized in the following theorem.

[3] Bounding the cost of *join*, *split*, and *join2* by the logarithm of the *smaller tree* is probably sufficient, but implementing a data structure with such bounds is very much more complicated.

Theorem 3.3.3. *For all strongly joinable trees (and treaps), the work and span of the algorithm (as shown in Figure 3.10) of* union, intersection *or* difference *on two balanced BSTs of sizes* m *and* n $(n \geq m)$ *is* $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ *(in expectation for treaps) and* $O(\log n \log m)$ *respectively (w.h.p. for treaps).*

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 $\log \binom{m+n}{n} = \Theta\left(m \log\left(\frac{n}{m}+1\right)\right)$ [164]. A generic proof of Theorem 3.3.3 working for all the four balancing schemes will be shown in the next section. The span of these algorithms can be reduced to $O(\log m)$ for weight-balanced trees even on the binary-forking model [81] by doing a more complicated divide-and-conquer strategy.

3.3.3 The Proof of Theorem 3.3.3

We now prove Theorem 3.3.3, for all the joinable trees and all three set algorithms (*union*, *intersection*, *difference*) from Figure 3.10.

For this purpose we make two observations. The first is that all the work for the algorithms can be accounted for within a constant factor by considering just the work done by the *split*s and the *join*s (or *join2*s), which we refer to as *split work* and *join work*, respectively. This is because the work done between each split and join is constant. The second observation is that the split work is identical among the three set algorithms. This is because the control flow of the three algorithms is the same on the way down the recursion when doing *split*s—the algorithms only differ in what they do at the base case and on the way up the recursion when they join.

Given these two observations, we prove the bounds on work by first showing that the join work is asymptotically at most as large as the split work (by showing that this is true at every node of the recursion for all three algorithms), and then showing that the split work for *union* (and hence the others) satisfies our claimed bounds.

We start with some notation, which is summarized in Table 3.2. In the three algorithms the first tree (T_1) is split by the keys in the second tree (T_2). We therefore call the first tree the *decomposed tree* and the second the *pivot tree*, denoted as T_d and T_p respectively. The tree that is returned is denoted as T. Since our proof works for either tree being larger, we use $m = \min(|T_p|, |T_d|)$ and $n = \max(|T_p|, |T_d|)$. We denote the subtree rooted at $v \in T_p$ as $T_p(v)$, and the tree of keys from T_d that v splits as $T_d(v)$ (i.e., *split* $(v, T_d(v))$ is called at some point in the algorithm). For $v \in T_p$, we refer to $|T_d(v)|$ as its *splitting size.*

[4] The nodes in $T_d(v)$ form a subset of T_d, but not necessarily a subtree. See details later.

Notation	Description
T_p	The pivot tree
T_d	The decomposed tree
n	$\max(\lvert T_p\rvert, \lvert T_d\rvert)$
m	$\min(\lvert T_p\rvert, \lvert T_d\rvert)$
$T_p(v), v \in T_p$	The subtree rooted at v in T_p
$T_d(v), v \in T_p$	The tree from T_d that v splits[4]
s_i	The number of nodes in layer i
v_{kj}	The j-th node on layer k in T_p
$d(v)$	The number of nodes attached to a layer root v in a treap

Table 3.2: Descriptions of notations used in the proof.

Figure 3.11 (a) illustrates the pivot tree with the splitting size annotated on each node. Since *split* has logarithmic work, we have,

$$\text{split work } = O\left(\sum_{v \in T_p} (\log |T_d(v)| + 1)\right),$$

which we analyze in Theorem 3.3.7. We first, however, show that the join work is bounded by the split work. We use the following Lemma.

Lemma 3.3.4. *For* T =union (T_p, T_d) *on strongly joinable trees, then* $\max(rank(T_p), rank(T_d)) \leq rank(T) \leq rank(T_p) + rank(T_d)$.

Proof. We prove it by induction on the tree size. For small trees, this conclusion obviously holds. Note that T_d will be split up into two trees T_l and T_r, with rank at most $r = rank(T_d)$ (Theorem 3.3.1). $lc(T_p)$ and $rc(T_p)$ will take *union* with either T_l or T_r, i.e., $L = union(lc(T_p), T_l)$ and $R = union(lc(T_p), T_l)$. Because of the induction hypothesis, $0 \leq rank(L) \leq rank(lc(T_p)) + rank(T_l) \leq rank(lc(T_p)) + r$, and similarly $0 \leq rank(R) \leq rank(lc(T_p)) + r$. From the **submodularity rule** joining them increase the rank of T_p by at least 0 and at most $rank(T_d)$. □

Theorem 3.3.5. *For each function call to* union, intersection *or* difference *on strongly joinable trees and treaps* $T_p(v)$ *and* $T_d(v)$*, the work to do the* join *(or* join2*) is asymptotically no more than the work to do the* split.

Proof. For *intersection* or *difference*, the cost of *join* (or *join2*) is $O(\log(|T|))$, where T is the result tree. Notice that *difference* returns the keys in $T_d \backslash T_p$. Thus for both *intersection* and *difference* we have $T \subseteq T_d$. The join work is $O(\log(|T|))$, which is no more than $O(\log(|T_d|))$ (the split work).

For *union*, if $rank(T_p) \leq rank(T_d)$, the *join* will cost $O(rank(T_d))$, which is no more than the split work.

Consider $rank(T_p) > rank(T_d)$ for strongly joinable trees. The rank of $lc(T_p)$ and $rc(T_p)$, which are used in the recursive calls, are at least $rank(T_p) - c_u$. Using Lemma 3.3.4, the rank of the two trees returned by the two recursive calls will be at least $(rank(T_p) - c_u)$ and at most $(rank(T_p) + rank(T_d))$, and differ by at most $O(rank(T_d)) = O(\log |T_d|)$. Thus the join cost is $O(\log |T_d|)$, which is asymptotically no more than the split work.

Consider $rank(T_p) > rank(T_d)$ for treaps. If $rank(T_p) > rank(T_d)$, then $|T_p| \geq |T_d|$. The root of T_p has the highest priority among all $|T_p|$ keys, so on expectation it takes at most the $\frac{|T_p|+|T_d|}{|T_p|} \leq 2$-th place among all the $|T_d| + |T_p|$ keys. From Lemma 3.2.8 we know that the cost on expectation is $\mathbb{E}[\log t] + 1 \leq \log \mathbb{E}[t] + 1 \leq \log 2 + 1$, which is a constant. □

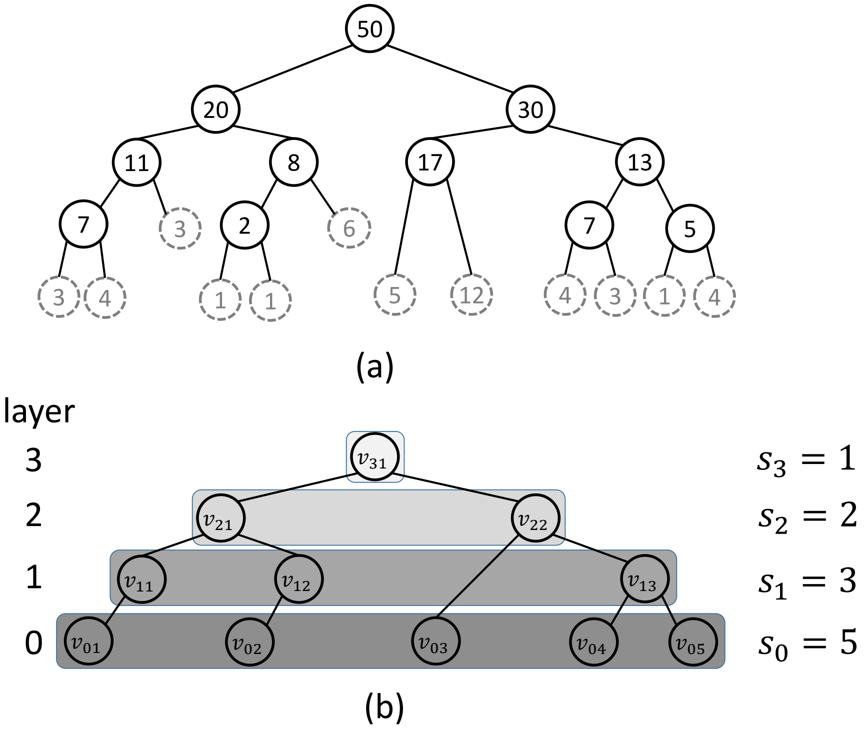


Figure 3.11: An illustration of splitting tree and layers – The tree in (a) is T_p, the dashed circle are the exterior nodes. The numbers on the nodes are the sizes of the tree from T_d to be split by this node, i.e., the "splitting size" $|T_d(v)|$. In (b) is an illustration of layers on an AVL tree.

This implies the total join work is asymptotically bounded by the split work.

We now analyze the split work. We do this by layering the pivot tree starting at the leaves and going to the root and such that nodes in a layer are not ancestors of each other.

We use the definition in Section 3.1. We define layers based on the ranks and denote the number of rank(i)-root nodes as s_i. Theorem 3.1.8 shows that s_i shrinks geometrically for joinable trees, which helps us prove our bound on the split work. Figure 3.11 (b) shows an example of the layers of an AVL tree on the two input trees of the *join*-based set-set functions.

Lemma 3.3.6. *For any ancestor-free set $V \subseteq T_p$, $\sum_{v \in V} |T_d(v)| \leq |T_d|$.*

The proof of this Lemma is straightforward.

Not all nodes are rank roots. However Theorem 3.1.7 shows that each rank cluster attached to a rank root contains only a constant number of nodes, and they are all the rank root's descendants.

By applying Theorem 3.1.8 and Theorem 3.1.7 we prove the split work. In the following proof, we denote v_{kj} as the j-th node in layer k.

Theorem 3.3.7. *The split work in* union, intersection *and* difference *on two joinable trees of size m and n is $O\left(m \log\left(\frac{n}{m} + 1\right)\right)$.*

Proof. The total work of *split* is the sum of the log of all the splitting sizes on the pivot tree $O\left(\sum_{v \in T_p} \log(|T_d(v)| + 1)\right)$. Denote l as the number of layers in the tree. Also, notice that in the pivot tree, in each layer there are at most $|T_d|$ nodes with $|T_d(v_{kj})| > 0$. Since those nodes with splitting sizes of 0 will not cost any work, we can assume $s_i \leq |T_d|$. We calculate the dominant term $\sum_{v \in T_p} \log(|T_d(v)| + 1)$ in the complexity by summing the work across layers. We first only consider all rank roots.

$$\begin{aligned}
\sum_{k=0}^{l} \sum_{j=1}^{s_k} \log\left(|T_d(v_{kj})| + 1\right) &\leq \sum_{k=0}^{l} s_k \log\left(\frac{\sum_j |T_d(v_{kj})| + 1}{s_k}\right) \\
&= \sum_{k=0}^{l} s_k \log\left(\frac{|T_d|}{s_k} + 1\right)
\end{aligned}$$

We split it into two cases. If $|T_d| \geq |T_p|$, $\frac{|T_d|}{s_k}$ always dominates 1. we have:

$$
\begin{align}
\sum_{k=0}^{l} s_k \log\left(\frac{|T_d|}{s_k} + 1\right) &= \sum_{k=0}^{l} s_k \log\left(\frac{n}{s_k} + 1\right) \tag{3.7} \\
&\leq \sum_{k=0}^{l} \frac{m}{2^{\lfloor k/c \rfloor}} \log\left(\frac{n}{m/c^{\lfloor k/c \rfloor}} + 1\right) \tag{3.8} \\
&\leq c \sum_{k=0}^{l/c} \frac{m}{2^k} \log \frac{n}{m/2^k} \notag \\
&\leq c \sum_{k=0}^{l/c} \frac{m}{2^k} \log \frac{n}{m} + 2 \sum_{k=0}^{l/c} k \frac{m}{2^k} \notag \\
&= O\left(m \log \frac{n}{m}\right) + O(m) \notag \\
&= O\left(m \log\left(\frac{n}{m} + 1\right)\right) \tag{3.9}
\end{align}
$$

If $|T_d| < |T_p|$, $\frac{|T_d|}{s_k}$ can be less than 1 when k is smaller, thus the sum should be divided into two parts. Also note that we only sum over the nodes with splitting size larger than 0. Even though there could be more than $|T_d|$ nodes in one layer in T_p, only $|T_d|$ of them should count. Thus we assume $s_k \leq |T_d|$, and we have:

$$
\begin{align}
\sum_{k=0}^{l} s_k \log\left(\frac{|T_d|}{s_k} + 1\right) &= \sum_{k=0}^{l} s_k \log\left(\frac{m}{s_k} + 1\right) \tag{3.10} \\
&\leq \sum_{k=0}^{2 \log_c \frac{n}{m}} |T_d| \log(1 + 1) \notag \\
&\quad + \sum_{k=c \log_2 \frac{n}{m}}^{l} \frac{n}{2^{\lfloor k/c \rfloor}} \log\left(\frac{m}{n/2^{\lfloor k/c \rfloor}} + 1\right) \tag{3.11} \\
&\leq O\left(m \log \frac{n}{m}\right) + c \sum_{k'=0}^{\frac{l}{c} - \log_2 \frac{m}{n}} \frac{m}{2^{k'}} \log 2^{k'} \notag \\
&= O\left(m \log \frac{n}{m}\right) + O(m) \notag \\
&= O\left(m \log\left(\frac{n}{m} + 1\right)\right) \tag{3.12}
\end{align}
$$

From (3.7) to (3.8) and (3.10) to (3.11) we apply Lemma 3.1.8 and the fact that $f(x) = x \log(\frac{n}{x} + 1)$ is monotonically increasing when $x \leq n$.

The above cost does not consider the nodes that are not rank roots. Recall that we use $d(v)$ to denote the rank cluster of a rank root v. Applying Theorem 3.1.7, the expectation is less than:

$$\begin{aligned} & \sum_{k=0}^{l}\sum_{j=1}^{x_k} d(v_{kj}) \log((T_d(v_{kj}) + 1) \\ = \ & d(v_{kj}) \times 2 \sum_{k=0}^{l}\sum_{j=1}^{x_k} \log((T_d(v_{kj}) + 1) \\ = \ & O\Big(m \log\Big(\frac{n}{m} + 1\Big)\Big) \end{aligned}$$

This holds in expectation for treaps.

To conclude, the split work on all four balancing schemes of all three functions is $O\big(m \log(\frac{n}{m} + 1)\big)$. □

Theorem 3.3.8. *The total work of* union, intersection *or* difference *of all four balancing schemes on two trees of size m and n (m ≥ n) is* $O\big(m \log(\frac{n}{m} + 1)\big)$.

This directly follows Theorem 3.3.5 and 3.3.7.

Theorem 3.3.9. *The span of* union *and* intersection *or* difference *on all four balancing schemes is* $O(\log n \log m)$*. Here n and m are the sizes of the two tree.*

Proof. For the span of these algorithms, we denote $D(h_1, h_2)$ as the span on *union*, *intersection* or *difference* on two trees of height h_1 and h_2. According to Theorem 3.3.5, the work (span) of *split* and *join* are both $O(\log |T_d|) = O(h(T_d))$. We have:

$$D(h(T_p), h(T_d)) \leq D(h(T_p) - 1, h(T_d)) + 2h(T_d)$$

Thus $D(h(T_p), h(T_d)) \leq 2h(T_p)h(T_d) = O(\log n \log m)$. □

Combine Theorem 3.3.8 and 3.3.9 we come to Theorem 3.3.3.

3.3.4 Other Tree algorithms Using *join*

Insert and Delete. Instead of the classic implementations of *insert* and *delete*, which are specific to the balancing scheme, we define versions based purely on *join*, and hence independent of the balancing scheme.

We present the pseudocode in Figure 3.12 to insert an entry e into a tree T. The base case is when t is empty, and the algorithm creates a new node for e. Otherwise, this algorithm compares k with the key at the root and recursively inserts e into the left or right subtree. After that, the two subtrees are *join*ed again using the root node. Because of the correctness of the *join* algorithm, even if there is imbalance, *join* will resolve the issue.

Build

```
build_sorted($S, i, j$) {
  if $i = j$ then return $\emptyset$;
  if $i + 1 = j$ then
    return singleton($S[i]$);
  $m = (i + j)/2$;
  $L$ = build'($S, i, m$) ||
    $R$ = build'($S, m + 1, j$);
  return join($L, S[m], R$); }
build($S, m$) {
  (S2, m2) = sort_rm_dup(S, m);
  build_sorted(S2,0,m2);}
```

Filter

```
filter($T, f$) {
  if $T = \emptyset$ then return $\emptyset$;
  ($L, e, R$) = expose($T$);
  $L'$ = filter($L, f$) ||
      $R'$ = filter($R, f$);
  if $f(e)$ then return join($L', e, R'$);
  else join2($L', R'$); }
```

Map and Reduce

```
map_reduce($T, g', f', I'$) {
  if $T = \emptyset$ then return $I'$;
  $\langle L, k, v, R\rangle$ = expose($T$);
  $L'$ = MapReduce($L, g', f', I'$) ||
      $R'$ = MapReduce($R, g', f', I'$);
  return $f'(L', f'(g'(k, v), R'))$; }
```

Range

```
range($T, l, r$) {
  ($T_1, T_2$) = split($T, l$);
  ($T_3, T_4$) = split($T_2, r$);
  return $T_3$; }
```

Foreach Index

```
foreach_index(T, $\phi$, s) {
  if (t = $\emptyset$) return;
  ($L, e, R$) = expose($T$);
  left = size(L);
  L = foreach_index(L, $\phi$, s); ||
    R = foreach_index(R, $\phi$, s+1+left);
  $\phi$(e, left);}
```

Insertion

```
insert($T, e$) {
  if $T = \emptyset$ then return singleton($e$);
  $\langle L, e', R\rangle$ = expose($T$);
  if $k(e) = k(e')$ then return T;
  if $k(e) < k(e')$ then return join(insert($L, e$), $e', R$);
  return join($L, e'$, insert($R, e$)); } }
```

Deletion

```
delete($T, k$) {
  if $T = \emptyset$ then return $\emptyset$;
  $\langle L, e', R\rangle$ = expose($T$);
  if $k < k(e')$ then return join(delete($L, k$), $e', R$);
  if $k(e') < k$ then return join($L, e'$, delete($R, k$));
  return join2($L, R$); }
```

Multi-insertion

```
multi_insert_s(T, A, m) {
  if (T = $\emptyset$) return build(A, m);
  if (m = 0) return t;
  $\langle L, e, R\rangle$ = expose($T$);
  b = binary_search(A, m, k(e));
  d = (b < m) and (k(A[b]) > k(e));
  L = multi_insert_s(r->lc, A, b) ||
    R = multi_insert_s(r->rc, A+b-d, m-b-d);
  return concat(L, e, R); }
multi_insert(t, A, m) {
  (A2, m2) = sort_rm_dup(A, m);
  return multi_insert_sorted(t, A2, m2);}
```

Figure 3.12: Pseudocode of some *join*-based functions – They are all independent of balancing schemes. The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

The *delete* algorithm is similar to *insert*, except when the key to be deleted is found at the root, where *delete* uses *join2* to connect the two subtrees instead. Both the *insert* and the *delete* algorithms run in $O(\log n)$ work (and span since sequential).

One might expect that abstracting insertion or deletion using *join* instead of specializing for a particular balance criteria has significant overhead. Our experiments show this is not the case—and even though we maintain the reference counter for persistence, we are only 17% slower sequentially than the highly-optimized C++ STL library (see section 11.6).

Theorem 3.3.10. *The* join*-based insertion algorithm cost time at most* $O(\log |T|)$ *on a joinable tree* T*. The rank of the output tree is at most* $c_u + rank(T)$*.*

Proof. We prove this by induction. This is obviously true for the base case. Consider $T = node(L, t, R)$, assume e goes to the left subtree L, and the new left subtree is L'. Then the output tree is $T' = join(L', t, R)$. Because of the induction hypothesis $rank(L') \leq rank(L) + c_u$. Because of the **submodularity rule**, $rank(T') \leq rank(T) + c_u$. □

Theorem 3.3.11. *For a joinable tree* T *with weight* $n = |T_1|$*, the* join*-based deletion algorithm cost time* $O(\log n)$*. The rank of the output tree is at most* $rank(T)$*.*

Proof. We first prove this by induction. This is obviously true for the base case. Consider $T = node(L, t, R)$, assume k falls in the right subtree R, and the new left subtree is R'. Then the output tree is $T' = join(L, t, R')$. From the induction hypothesis, $rank(R') \leq rank(R)$. Based on the decreasing side of the **submodularity rule**, $rank(T') \leq rank(T)$. This proves that the output tree of *delete* is at most the rank of the input.

In *delete*, there is at most one invocation of *join2*, which takes time no more than $\log n$. For $T' = join(L, t, R')$, we next prove that the cost. Consider two cases.

1. The key $k \neq k(R)$. WLOG assume k falls in the right subtree of R (the other case is symmetric). R' =*join* $(lc(R), R, R_r)$ where R_r =*delete* $(rc(R), k)$. Then

$$rank(R') \geq rank(lc(R)) \geq rank(R) - c_u \geq rank(L) - 2c_u + c_l$$

   Meanwhile $rank(R') \leq rank(R)$ as we have proved above. Therefore

$$rank(R') \leq rank(R) \leq rank(L) + c_u - c_l$$

   In summary $rank(L)$ and $rank(R')$ differs by a constant. The cost of a single *join* is a constant.

2. The key $k = k(R)$. R' =*join2* $(lc(R), rc(R))$. The cost of this *join* is at most $O(\log n)$, but this only happens once.

In summary, the total cost of all *join* is $h(T) + O(\log n) = O(\log n)$, and the cost of the *join2* is at most $O(\log n)$. This proves the above theorem. □

Build. A balanced binary tree can be created from a sorted array of key-value pairs using a balanced divide-and-conquer over the input array and combining with *join*. To construct a balanced binary tree from an arbitrary array we first sort the array by the keys, then remove the duplicates. All entries with the same key are consecutive after sorting, so the algorithm first applies a parallel sorting and then follows by a parallel packing. The algorithm then extracts the median in the de-duplicated array, and recursively construct the left/right subtree from the left/right part of the array, respectively. Finally, the algorithm uses *join* to connect the median and the two subtrees. The work is then $O(W_{\text{sort}(n)}+W_{\text{remove}(n)}+n)$ and the span is $O(S_{\text{sort}(n)} + S_{\text{remove}(n)} + \log n)$. For work-efficient sort and remove-duplicates algorithms with $O(\log n)$ span this gives the bounds in Table 3.1.

Bulk Updates. We use *multi_insert* and *multi_delete* to commit a batch of write operations. The function *multi_insert*(T, A, m) takes as input a P-Tree root t, and the head pointer of an array A with its length m.

We present the pseudocode of *multi_insert* in Figure 3.12. This algorithm first sorts A by keys, and then removes duplicates in a similar way as in *build*. We then use a divide-and-conquer algorithm *multi_insert_s* to insert the sorted array into the tree. The base case is when either the array A or T is empty. Otherwise, the algorithm uses a binary search to locate t's key in the array, getting the corresponding index b in A. d is a bit denoting if k appears in A. Then the algorithm recursively multi-inserts A's left part (up to $A[b]$) into the left subtree, and A's right part into the right subtree. The two recursive calls can run in parallel. The algorithm finally concatenates the two results by the root of T. A similar divide-and-conquer algorithm can be used for *multi_delete*, using *join2* instead of *join* when necessary.

Decoupling sorting from inserting has several benefits. First, parallel sorting is well-studied and there exist highly-optimized sorting algorithms that can be used. This simplifies the problem. Second, after sorting, all entries in A that to be merged with a certain subtree in T become consecutive. This enables the divide-and-conquer approach which provides good parallelism, and also gives better locality.

The total work and span of inserting or deletion an array of length m into a tree of size $n \geq m$ is $O\big(m \log\big(\frac{n}{m} + 1\big)\big)$ and $O(\log m \log n)$, respectively [74]. The analysis is similar to the *union* algorithm.

Range. *range* extracts a subset of tuples in a certain key range from a P-Tree, and output them in a new P-Tree. The cost of the *range* function is $O(\log n)$. The pure *range* algorithm copies nodes on two paths, one to each end of the range, and using them as pivots to *join* the subtrees back. When the extracted range is large, this pure *range* algorithm is much more efficient (logarithmic time) than visiting the whole range and copying it.

Filter. The *filter*(t, ϕ) function returns a tree with all tuples in T satisfying a predicate ϕ. This algorithm filters the two subtrees recursively, in parallel, and then determines if the root satisfies ϕ. If so, the algorithm uses the root as the pivot to *join* the two recursive results. Otherwise it calls *join2*. The work of *filter* is $O(n)$ and the depth is $O(\log^2 n)$ where n is the tree size.

Map and Reduce. The function *map_reduce*$(T, f_m, \langle f_r, I\rangle)$ on a tree t (with data type E for the tuples) takes three arguments and returns a value of type V'. $f_m : E \mapsto V'$ is the a map function that converts each stored tuple to a value of type V'. $\langle f_r, I\rangle$ is a monoid where $f_r : V' \times V' \mapsto V'$ is an associative reduce function on V', and $I \in V'$ is the identity of f_r. The algorithm will recursively call the function on its two subtrees in parallel, and reduce the results by f_r afterwards.

3.3.5 Extend the Algorithms for Combining Values

For supporting key-value pairs as entries, this thesis further extend some functions to accept a complementary function $\sigma : V \times V \mapsto V$ as an argument, where V is the value type in the tree. 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 *intersection*, 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 *intersection* sum the counts, *insert* add the counts, and *build* keep the total counts of the same key in the sequence. We will see a use example of this complementary function in counting queries in 2D range, segment or rectangle queries in Section 9. Another simple use example is to implement an *update* or *multi_update* operation by calling *insert* or *multi_insert*, using the complementary function as *replace* : $(a, b) \mapsto b$. This means to usually keep the later value when duplicate keys appear.

Accordingly, in the algorithms, we will need to update the values when applicable. For example, in Line 4 in the *insert* algorithm, when the key to be inserted already exists, we directly update the value of the root. For *union* and *multi_insert* this means to update the value of the pivot of *join*, if the corresponding key appears in both input sets, no matter in the tree or the array. For *build* and *multi_insert*, it requires the preprocessing sorting to also combine values of duplicate keys (instead of just remove duplicates) in the input array. This is still easy using a parallel packing since the duplicate keys are contiguous after sorting.

3.4 Discussion of the *join*-based Algorithms

Abstracting all balancing property into *join* has been studied in other previous work. (more information)

Interestingly, combining it with *expose*, the *join*-based algorithms even have nothing specific to do with the tree structure. We define *expose* (S) on an ordered set (or even a sequence) as to return two subsets of it L and R, and an entry $e \in S$, where $[L, e, R] = S$. For trees, *expose* naturally makes use of the two subtrees and the root. It is also applicable to arrays or lined lists, where it returns e as a middle entry and breaks the input into its left and right halves separated by e. We also define a more general *join* (L, e, R) as combining two ordered sets and an entry e into an order set $L \cup \{e\} \cup R$, where $\max(L) < e < \min(R)$.

Then all algorithms in Figures 3.8, 3.10, and 3.12 are still correct for any data structures. However they are not necessary to be efficient. For some efficiency, we would require both *expose* and *join* to take logarithmic time, and L and R returned by *expose* to be of similar ranks. Binary trees satisfy these conditions, but arrays may not be sufficient. This would also leads to a combine function for skip lists, also in parallel.

Previous work has also made attempt to unify multiple balancing schemes, and some of them also use rank [150, 153]. Haeupler et al. define ranks for several commonly-used balancing schemes, and define rank-based trees [153]. Our framework is also applicable to their weak AVL trees (or WAVL trees) since the authors show the correspondence of WAVL to RB trees [153]. However they only consider height-balanced trees and they did not discuss bulk operations on trees.

Why We Need a Unified Framework for Multiple Balancing Schemes? Although *join* unifies multiple balancing schemes, it is worth discussing why it is useful and meaningful for algorithm design. These balancing schemes were invented based on different backgrounds, and thus demonstrate different properties in different applications. For example, in geometry queries, such as a range tree (see Section 9.3.1), using WB trees can bound the amortized bound of an insertion by a logarithmic time [195], while AVL or RB trees need linear time in the worst case. However, when write-efficiency is preferred, the AVL trees and treaps provide the better performance because they require less writes to data (see details in Chapter 14). Also, some balancing schemes has been used in existing systems and libraries, e.g., the STL uses red-black trees for implementing ordered sets. Having generic approaches over different balancing schemes makes it easier to adopt the algorithms to different scenarios, without breaking the original restrictions. Secondly, it is out of our intellectual curiosity to study the common properties of different balancing schemes, and to abstract out rules that enables balance efficiency in algorithms. This enhances our understanding to balanced binary tree structures and the essential similarities and differences between balancing schemes.

Chapter 4

Augmenting Trees

4.1 Augmented Trees

To allow for more functionalities on trees, especially some aggregation operations, we usually *augment* the tree. A generalized 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 [117], 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 as binary search trees with respect to ordered maps and an associative reduce operation. The data stored within each tree node is an entry e of a key-value pair with key type K and value type V. Tree nodes are sorted by keys. Each node also keeps track of the *augmented value* of the sub-tree rooted at it. These augmented values are partial sums keeping track of some aggregation of the whole subtree. These partial sums are especially useful in some range sum queries such that the query does not need to scan all data in the range. When analyzing the data for certain trends, it is likely useful to quickly query the sum or maximum of value within a key range. More generally the augmentation can be used for quick interval queries, k-dimensional range queries, inverted indices (all described later in the thesis), segment intersection, windowing queries, point location, rectangle intersection, range overlaps, and many others. More general than just the sum or the maximum value, these augmented values can be defined more generally using an augmenting structure, which is a map-reduce-like operation.

Definition 6 (Augmenting Structure)**.** *An augmenting structure about an key-value entry type $K \times V$ is a tuple $aug_{K,V} = \langle A, g, f, a_\emptyset \rangle$, where the parameters are:*

A,	*The augmented value type*
$g : K \times V \rightarrow A$,	*The base function*
$f : A \times A \rightarrow A$,	*The combine function*
$a_\emptyset : A$,	*Identity for f*

We omit the subscript when the context is clear.

In other words, an augmented tree associates a plain search tree with an augmenting structure *aug* for supporting quick "sum" operations over a range of keys, where *sum* means with respect to any associative combine function. An augmenting structure consists 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 the identity of f, $a_\emptyset$. The augmenting structure defines a map-reduce-like operation on the entries in the tree. In particular, the base function maps an entry (a key-value pair) to an augmented value. This function decides the augmented value of a single non-empty tree node. The combine function f reduces all such augmented values of individual entries, which can be used to compute the augmented value of a whole subtree. This function is required to be associated. The identity of f, $a_\emptyset$, intuitively is the augmented value of an empty tree. $(A, f, a_\emptyset)$ is a monoid. The *augmented value* is then the "sum" obtained by using the augmenting structure, which is defined as:

Definition 7 (Augmented Value of a Subtree). *Given $t = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$, which is a search tree storing key-value pairs $\{(k_i, v_i)\}$ ordered by the tree symmetric order, and an augmenting structure aug $= \langle A, g, f, a_\emptyset \rangle$. The augmented value of the tree is defined as*

$$\mathcal{A}_{\langle A,g,f,a_\emptyset \rangle}(t) = f(g(k_1, v_1), g(k_2, v_2), \ldots, g(k_n, v_n)) \tag{4.1}$$

We will omit the subscript when the context is clear. For the binary associative function f, we define:

$$f(\emptyset) = a_\emptyset \tag{4.2}$$
$$f(a_1) = a_1 \tag{4.3}$$
$$f(a_1, a_2, \ldots, a_n) = f(a_1, f(a_2, \ldots, a_n)) \tag{4.4}$$

Therefore, the we formally define *augmented tree* as follows:

Definition 8 (Augmented Tree). *An augmented tree type $T = \mathbb{AT}(K, <_K, V, aug)$ or $T = \mathbb{AT}(K, <_K, V, A, g, f, a_\emptyset)$ is a binary search tree where each tree node stores a data entry and an augmented value of its subtree. In particular, the data are key-value pairs $\in K \times V$, with the tree symmetric order on K defined by $<_K$. The augmented values are defined by the augmenting structure aug such that each tree node u stores the augmented value $\mathcal{A}_{aug}(T_u)$.*

The augmenting structure is chosen ahead of time, based on the applications. For the same dataset, different augmentations (i.e., different augmenting functions) can lead to different functionality.

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

$\mathcal{A}(u) = f(\mathcal{A}(lc(u)), g(k(u), v(u)), \mathcal{A}(rc(u)))$. Considering the associativity of the combine function f, as well as the ordering of search trees, the augmented value stored in each node of an augmented tree is unique, regardless of the shape or balanceness of the tree structure, and is exact the augmented value of the map containing all entries in the tree. The same argument holds for each subtree.

The augmented trees are especially useful in reporting range queries. As an example of such a range sum consider a database of sales receipts keeping the value of each sale ordered by the time of sale. When analyzing the data for certain trends, it is likely useful to quickly query the sum or maximum of sales during a period of time. Although such sums can be implemented naively by scanning and summing the values within the key range, these queries can be answered much more efficiently using augmented trees [117, 134], using the augmentation of the sum of sale value in each subtree. The sum of any range on a tree of size n can be answered in $O(\log n)$ time. Similar approach can be used to answer the maximum value in a period and so on.

4.2 Algorithms on Augmented Trees

As a special search tree structure, all the tree algorithms in Section 3.3 are applicable also on augmented trees. The only difference is that the input and the corresponding output tree structures should have valid augmented values.

As mentioned, the augmented value of a tree node u depends on the entry in this node, and the augmented value of its left and right children. Accordingly, we need to update $\mathcal{A}(u)$ when any update overlaps u's subtree. For example, when u is involved in rotations, or when u itself is updated. In fact, all such scenarios happen only when we call $node(L, u, R)$ to connect u with its left and right children. This include when we create a new node $u = node(\emptyset, u, \emptyset)$. All such invocations happen only in the *join* algorithm. As a result, to make the general tree algorithms to work on augmented trees, the only thing involved is to let the *join* algorithm re-compute the augmented values of the affected tree nodes. All the other algorithms are oblivious to the augmentation, and the implementation (and the code) is unaffected by adding any augmentation. When f and g both only take constant time, the asymptotic cost of all these tree algorithms also remain.

In this section, we further introduce some algorithms that are specific to augmented trees. The pseudocode of some algorithms in this section is presented in Figure 4.1.

aug_left, aug_right and aug_range. We define these three functions to extract the augmented value in a tree in a certain range. In particular, $aug\_left(T, k)$ and $aug\_right(T, k)$ means to return the augmented value of all entries with keys no greater (or no smaller) than k. $aug\_range(T, k_1, k_2)$ means to return the augmented value of all entries in key range from k_1 to k_2. Figure 4.1 shows the *aug_left* as an example. The other two functions are similar. This algorithm is similar to the *range* algorithm, which finds the path of the splitter key, and combine the relevant subtrees and single nodes—for any subtree in the

Aug-left

```
aug_left($T, k$) {
  $a$ = $a_\emptyset$;
  $T_0 = T$;
  while $T_0 \neq \emptyset$ {
    $(L, e, R)$ = expose($T_0$);
    if ($k \geq k(e)$) {
      $a$ = $f(a, \mathcal{A}(L), g(e))$;
      if ($k = k(e)$) break;
      $T_0$ = $R$;
    } else $T_0$ = $L$;}
  return $a$; }
```

Aug-filter

```
aug_filter($T, h$) {
  $(L, e, R)$ = expose($T$);
  if (not $h(\mathcal{A}(T))$) return $\emptyset$;
  $L'$ = aug_filter($L, h$) || $R'$ = aug_filter($R, h$);
  if ($h(g(e))$) return join($L', e, R'$);
  else return join2($L', R'$);}
```

Aug-project

```
aug_project_left(T,g’,f’,k) {
  $a$ = $g'(a_\emptyset)$; $T_0 = T$;
  while $T_0 \neq \emptyset$ {
    $(L, e, R)$ = expose($T_0$);
    if ($k \geq k(e)$) {
      $a$ = $f'(a, g'(\mathcal{A}(L)), g'(g(e)))$;
      if ($k = k(e)$) break;
      $T_0$ = $R$;
    } else $T_0$ = $L$;}
  return $a$; }

aug_project($T, g', f', k_1, k_2$) {
  $T_0$ = $T$; $(L, e, R)$ = expose($T_0$);
  while ($k_2 < k(e)$ || $k_1 > k(e)$) {
    if ($k_2 < k$) $T_0$ = $L$; else $T_0$ = $R$;
    if ($T_0$ = $\emptyset$) break;
    $(L, e, R)$ = expose($T_0$); }
  if ($T_0$ = $\emptyset$) return $g'(a_\emptyset)$;
  $l$ = aug_project_right(L,g’,f’,k_1);
  $r$ = aug_project_left(R,g’,f’,k_2);
  return $f'(l, g'(g(e)), r)$; } }
```

Figure 4.1: Pseudocode of some algorithms on augmented tree – The syntax $S_1||S_2$ means that the two statements S_1 and S_2 can be run in parallel based on any fork-join parallelism.

range, we add its augmented value, and for a single node, we add the value of applying g on its entry. All such partial augmented values are combined by f. Because of the augmented values stored in each tree node, this only requires $O(\log n)$ times of application of f and g for a tree of size n.

aug_filter. The *aug_filter* function aims at using augmentation to accelerate some special *filter* operation. The motivation of using augmentation for a *filter* function is slightly different from reporting range sums. The idea is to use the augmented value as an indicator if the whole subtree should be kept or dropped. The *augFilter*(T, h) function is equivalent to *filter*(T, h'), where $h' : K \times V \mapsto$ *Bool* satisfies $h(g(k, v)) \Leftrightarrow h'(k, v)$. It is only applicable if $h(a) \vee h(b) \Leftrightarrow h(f(a, b))$ for any a and b ($\vee$ is the logical or). In this case the *filter* function can make use of the augmented values such that when the augmented value of a whole subtree does not satisfy h, the whole subtree can be discarded. For example, assume the values in the map are boolean values, f is a logical-or, $g(k, v) = v$, and we want to filter the map using function $h'(k, v) = v$. In this case we can filter out a whole sub-map once we see it has *false* as a partial sum. Hence we can set $h(a) = a$ and directly use *aug_filter* (T, h). In our example of sales receipts over time with augmentation to be the maximum sale, this function can be used to filter out all sales above a given amount very much more efficiently than scanning a whole range of sales. The function is used in interval trees (Chapter 8). In fact, when the output size is k, the cost of *aug_filter* is $O(k \log(n/k + 1))$.

The *aug_filter* as introduced above aims at filtering *out* a whole subtree by taking advantage of the augmented values. Similarly, we can try to filter *in* a whole subtree by a different input predicator h_2. The *augFilter2*(T, h_2) function is equivalent to *filter*(T, h'), where $h' : K \times V \mapsto Bool$ satisfies $h(g(k, v)) \Leftrightarrow h'(k, v)$. It is only applicable if $h_2(a) \wedge h_2(b) \Leftrightarrow h_2(f(a, b))$ for any a and b ($\wedge$ is the logical and). These two versions of *aug_filter* can also be combined.

Theorem 4.2.1. *The* aug_filter *algorithm as shown in Figure 4.1 costs work $O(k \log(n/k+1))$ on the four joinable trees in this thesis, where k is the output tree size, if the input predicator function h costs constant time.*

We will prove this theorem in Section 4.3.

aug_project. The *augProject*(T, g', f', k_1, k_2) function is equivalent to $g'($*augRange*$(T, k_1, k_2))$. 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 is useful when the augmented values are themselves maps or other large data structures. It allows projecting the augmented values down onto another type by g' (e.g. project augmented values with complicated structures to values like their sizes) then summing them up by f', and is much more efficient when applicable. This is to avoid invoking the combining function f on trees when f is expensive. We can compute $g'($*aug_range*$(T, k_1, k_2))$ by just using g, g' and f'. Similarly we extract all the relevant subtrees and single tree nodes in the range of k_1 to k_2 in the tree. For each subtree, we apply g' on its augmented value; for each single node, we apply $g'(g(\cdot))$ on its entry. Then all these results will be reduced by f'. For example in range trees where each augmented value is itself an augmented map, it greatly improves performance for queries.

4.3 Proof of Theorem 4.2.1

To prove Theorem 4.2.1, we first present a useful lemma.

Lemma 4.3.1. *Suppose a tree T of size n satisfies that for any subtree $T_x \in T$, the height of T_x is $O(\log |T_x|)$. Let S be the set of all the ancestors of $m \leq n$ elements in T, then $|S| \in O(m \log(\frac{n}{m} + 1))$.*

Proof of Lemma 4.3.1. First of all, we find all the searching paths to all the m elements, and mark all related nodes on the path as red. All the red nodes form a tree structure, which is a connected component of T. We then adjust the red nodes, such that the number of red nodes does not decrease. We define a red node with two red children as a *joint* node, and a red node with one red children as a *linking* node.

1. First, as shown in Figure 4.2 (a), any of the red nodes are internal nodes in T, we arbitrarily extend it to some external node in the tree.
2. Second, as shown in Figure 4.2 (b), if there is any linking node v with some joint nodes as its descendent, then we move the first joint node of its descendants up to replace the non-red child of v.

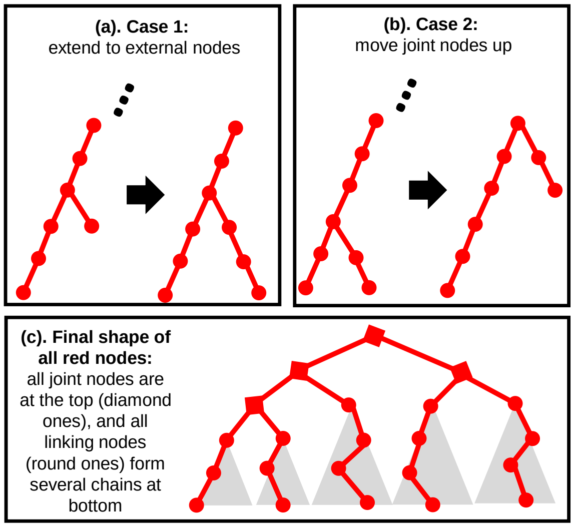

(a). Case 1:
extend to external nodes
(b). Case 2:
move joint nodes up
(c). Final shape of all red nodes:
all joint nodes are at the top (diamond ones), and all linking nodes (round ones) form several chains at bottom

Figure 4.2: An illustration about adjusting red nodes in the proof of Lemma 4.3.1 – (a) Extend all inner joint nodes to some external node. (b) Move all joint nodes to upper levels as far as possible. (c) The final shape after adjusting the red nodes. All joint nodes are at the top, and all linking nodes form chains at bottom.

We repeat the two steps until there is no such situations. These adjustments only make the total number of red nodes larger. Finally we will have all joint nodes on the top levels of the tree, forming a connected component. All the linking nodes form several (at most m) chains at the bottom levels. The total number of joint nodes is $O(m)$ because the number of chains is at most m. For all the linking nodes, note that the length of each chain corresponds to a subtrees in T, and all such subtrees are ancestor-free. We assume the size of the i-th subtree is n_i, then we have $\sum_{i=1}^{m} n_i = n$. The total length of all chains is:

$$\sum_{i=1}^{m} \log(n_i + 1) \leq m \log\left(\frac{\sum_{i=1}^{m} n_i + 1}{m}\right) \leq m \log\left(\frac{n}{m} + 1\right)$$

□

A more general version of Theorem 4.3.1 is proved in [81].

Now we can prove Theorem 4.2.1.

Proof of Theorem 4.2.1. There are two types of nodes visited by the algorithm in Figure 4.1: those that are visited but skipped in Line 3 (noted as *skipped nodes*) and those that are visited because at least one of its descender satisfies h (noted as *passing nodes*). Suppose h costs constant time, the cost is proportional to the total number of skipped nodes and passing nodes. The parent of a skipping node must be a passing node, so we charge the cost (only a constant) of visiting a skipping node to its parent, which does not affect the asymptotical cost. There are in total k nodes satisfying h. According to Theorem 4.3.1, the number of all their ancestors, which are all the passing nodes, is $O(m \log(\frac{n}{m} + 1))$. □

4.4 Augmented Maps

The augmented tree defined in Section 4.1 represents a special type of ordered maps (key-value store), which associates the map an *augmented value.* This thesis further propose an abstract data type *augmented maps* to represent this ordered map, which quickly return the (partial) augmented value of the map or a sub-map. Similar as the definition of augmented trees, we formally define the augmented maps as follows.

Definition 9 (Augmented Map). *An augmented map type* $\mathbb{AM}(K, <_K, V, aug = \langle A, g, f, a_\emptyset \rangle)$ *is an ordered map associated with an augmenting structure* $aug = \langle A, g, f, a_\emptyset \rangle$*, hence parameterized on the following seven parameters:*

$$\begin{array}{lll} K, & & \textit{Key type} \\ <_K & : K \times K \to \text{Bool}, & \textit{Total ordering on the keys} \\ V, & & \textit{Value type} \\ & & \\ A, & & \textit{Augmented value type} \\ g : K \times V \to A, & & \textit{The base function} \\ f : A \times A \to A, & & \textit{The combine function} \\ a_\emptyset : A, & & \textit{Identity for } f \end{array}$$

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). These functions are chosen ahead of time, based on the applications. For the same dataset, different augmentations (i.e., different augmenting functions) can lead to different functionality.

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

Definition 10 (Augmented Value). *Given* $m = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$*, which is an augmented map* $\mathbb{AM}(K, <_K, V, aug = \langle A, g, f, a_\emptyset \rangle)$*, its* augmented value *is*

$$\mathcal{A}(m) = f(g(k_1, v_1), g(k_2, v_2), \ldots, g(k_n, v_n)) \tag{4.5}$$

Equivalently, the augmented value of an augmented map $\mathbb{AM}(K, <_K, V, aug = \langle A, f, g, I \rangle)$ can be define recursively as:

$$\begin{array}{rcl} \mathcal{A}(\emptyset) & = & a_\emptyset \\ \mathcal{A}((k_1, v_1) :: m') & = & f(g(k_1, v_1), \mathcal{A}(m')) \end{array}$$

where $(k_1, v_1) :: m'$ means separating the input ordered map into its first (smallest as defined by $<_K$) entry (k_1, v_1) and the rest m'. It is also equivalent to extracting the last element and the rest since f is associative. The ordering matters since f need not be commutative.

As an example, the augmented map type for summing sales in ranges of time periods can be defined as:

$$\mathbb{AM}(\mathbb{Z}, <_\mathbb{Z}, \mathbb{R}, \mathbb{R}, (k, v) \to v, +_\mathbb{R}, 0) \tag{4.6}$$

It stores in each entry the time stamp ($\in \mathbb{Z}$) as keys, the sale amount ($\in \mathbb{R}$) at that time as values, ordered by $<_{\mathbb{Z}}$. The augmented value keeps track of the sum of its values (sales).

Usually, the augmenting structure is selected based on the application and desired functionality. In various applications, the key, value or augmented value type can be much more complicated types. For example, in the 2D range query (see Chapter 9.3), the augmented values are themselves augmented maps, making the formalization of the problem a nested augmented map.

Data Structures for Augmented Maps. Augmented maps are independent of representation. They can be implemented, for example, using sorted arrays, and for applications that do not have dynamic updates this would be a reasonable implementation. However, an efficient implementation using sorted arrays would still require storing partial sums of augmented values in a static tree. In fact, by maintaining such augmented values of sub-maps (partial sums) in the corresponding underlying data structures, we can answer these range-sum queries more efficiently.

As mentioned, one possible and efficient implementation is to use augmented trees [117, 134] storing partial sums in subtrees. The advantage of using search trees to implement augmented maps is that it returns range-related augmented value queries quickly, and is appropriate for the dynamic setting. Later in this chapter, we will show another possible implementation of augmented maps, the prefix structures, which is more efficient and practical in returning prefix-related augmented value queries. In the sales receipt example, both augmented trees and prefix structures can answer the sale sum of any time period for a total of n time units in $O(\log n)$ time. For trees, this bound is achieved by using a balanced binary tree and augmenting each node augmented with the sum of the subtree. This can be built upon the *join*-based algorithms, and experiments show that the additional cost for maintaining the augmentation is reasonably small (typically around 10% for simple augmenting functions such as summing the values or taking the maximum). For prefix structures, this requires that the combine function f ($+_{\mathbb{R}}$ in this case) have an applicable inverse function (accordingly $-_{\mathbb{R}}$).

4.5 Prefix Structures

In addition to using augmented trees to implement augmented maps, this thesis also propose *prefix structures* as another possible data structure for augmented maps. It is inspired from the sweepline algorithms for geometric problems, which uses a series of data structures to organize some information about all prefixes of the input data points. Similarly, for an augmented map $m = \{e_1, \ldots, e_{|m|}\}$, the prefix structures store the augmented values of all prefixes up to each entry e_i, i.e., $aug\_left(m, k(e_i))$. For example, if the augmented value is the sum of values, the prefix structures are prefix sums. We denote the prefix sum at entry i as t_i.

Build Prefix Structures in Parallel. Constructing the prefix structures sequentially is straight-forward as computing each prefix structure based on the previous one. In other words:

$$t_i = f(t_{i-1}, g(e_i))$$

In this thesis, we further present a parallel algorithm to build the prefix structures.

Sequentially, to compute a prefix structure t_i from a previous prefix structure t_j ($j < i$) means to repeatedly "add" a sequence of entries $\langle e_{j..i} \rangle$ onto t_j using the combine function f. A useful observation is that, because of the associativity of the combine function f, the aforementioned process is equivalent to directly combining the "partial sum", or the augmented value, of all entries $\langle e_{j..i} \rangle$ to t_i using f. Thus, to build all the prefix structures in parallel, our approach is to evenly split the input sequence of points into b blocks, calculate the partial sum of each block, and refine the prefix structures in each block using the update function h. For simplicity we assume n is an integral multiple of b and $n = b \times r$. We define a *fold function* $\rho_{f,g} : \langle P \rangle \mapsto T$ that converts a sequence of points into a prefix structure, which gives the "partial sums" of the blocks. We also define an *update function* $h_{f,g} : A \times E \mapsto A$, which adds an entry directly to an augmented value. It is used for refining the prefix structures inside each block. We note that both the fold function and the update function can be directly computed using g and f as:

$$\boldsymbol{\rho}_{\boldsymbol{f},\boldsymbol{g}}(p_1, \ldots p_n) \equiv f(g(p_1), \ldots g(p_n)) \tag{4.7}$$

$$\boldsymbol{h}_{\boldsymbol{f},\boldsymbol{g}}(a, e) \equiv f(a, g(e)) \tag{4.8}$$

When context is clear, we omit the subscript and denote them directly as ρ and h. Although we can always compute ρ and h using the original augmented map parameters g and f, in many applications, much simpler and efficient(sometimes also parallel) algorithms can be used for $\rho_{f,g}$ and $h_{f,g}$.

As a result, the parallel paradigm to generate the prefix structures of an augmented map $\mathbb{AM}(K, <_K, V, aug = \langle A, f, g, I \rangle)$ can be defined as follows:

$$\mathbf{S}' = \mathbb{PX}(\mathbf{K}; \quad \mathbf{<_K}; \quad \mathbf{V}; \quad \mathbf{A}; \quad \mathbf{a}_{\boldsymbol{\emptyset}}; \quad \mathbf{h}; \quad \boldsymbol{\rho}; \quad \mathbf{f}) \tag{4.9}$$

We do not include g in the formalization since it is never used separately. Our parallel algorithm to build the prefix structures is as follows (also see Algorithm 1 and Figure 4.3):

1. **Batching**. Assume all input entries have been sorted by $<_K$. We evenly split them into b blocks and then in parallel generate b partial sums $t'_i \in T$ using ρ, each corresponding to one of the b blocks. They are the augmented values (partial sums) of each block.
2. **Sweeping**. These partial sums t'_i are combined in turn sequentially by the combine function f to get the first prefix structure $t_0, t_r, t_{2r} \ldots$ in each block using $t_{i \times r} = f(t_{(i-1) \times r}, t'_i)$.

Algorithm 1: The construction of the prefix structure.

```
Input: A list $p$ storing $n$ input points in order, the update function $h$, the fold function $\rho$,
       the combine function $f_h$, the empty prefix structure $t_0$, and the number of blocks $b$.
       Assume $r = n/b$ is an integer.
Output: A series of prefix structure $t_i$.
[Step 1.] parallel-for $i \leftarrow 0$ to $b-1$ do
    $t'_i = \rho(p_{i\times r}, \ldots, p_{(i+1)\times r-1})$
[Step 2.] for $i \leftarrow 1$ to $b-1$ do  $t_{i\times r} = f_h(t_{(i-1)\times r}, t'_{i-1})$
[Step 3.] parallel-for $i \leftarrow 0$ to $b-1$ do
    $s = i \times r$, $e = s + r - 1$
    for $j \leftarrow s$ to $e$ do  $t_j = h(t_{j-1}, p_j)$
return $\{p_i \mapsto t_i\}$
```

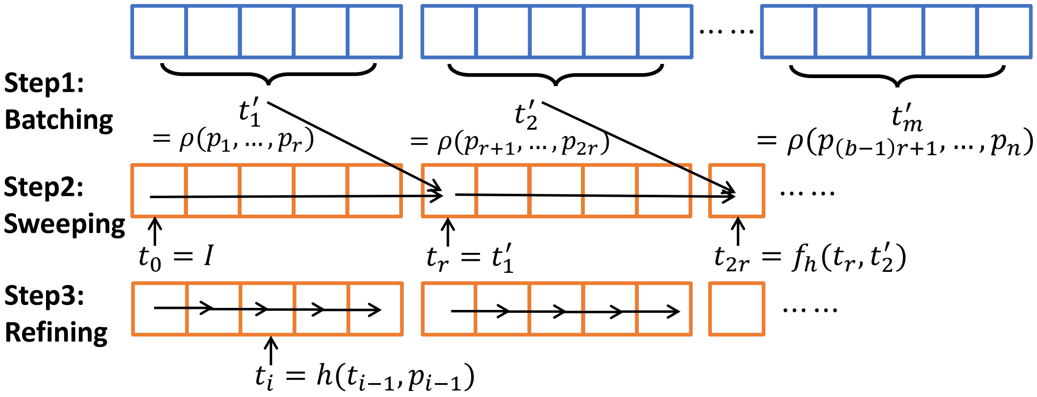

Step1:
Batching
t'_1
$= \rho(p_1, \ldots, p_r)$
t'_2
$= \rho(p_{r+1}, \ldots, p_{2r})$
……
t'_m
$= \rho(p_{(b-1)r+1}, \ldots, p_n)$
Step2:
Sweeping
……
$t_0 = I$
$t_r = t'_1$
$t_{2r} = f_h(t_r, t'_2)$
Step3:
Refining
……
$t_i = h(t_{i-1}, p_{i-1})$

Figure 4.3: The construction of the prefix structures.

3. **Refining**. All other prefix structures are built based on $t_0, t_r, t_{2r}, \ldots$ (built above in the second step) in the corresponding blocks. All the b blocks can be processed in parallel. In each block, the prefix structure t_i is computed sequentially in turn by applying h on e_i and t_{i-1}.

Algorithm 1 is straight-forward, yielding a simple implementation. Our experiments show that it also has good performance in parallel.

Theoretically, by repeatedly applying this process to each block in the last step, we can further enable more parallelism. The cost of the algorithm depends on the cost of the involved functions ρ, h and f. For simple augmentation, such as summing up the values, this algorithm is equivalent to computing all prefix sums in parallel. By using b as any constant, ρ as a parallel summation algorithm, and recursively applying this paradigm in Step 3, we can achieve the standard parallel prefix sum algorithm, which cost $O(n)$ work and $O(\log n)$ depth.

However, for more complicated augmentations, we note that the parallelism is non-trivial. For example, in the geometric problems as will be presented in Chapter 9, the formalization of prefix structures leads to sweepline algorithms. However, the augmentation are more complicated, as the augmented values in these applications are themselves maps (trees), and the combine functions are set operations such as a *union*. The challenge of constructing these prefix structures is in the sweeping step where f is applied sequentially for b times, which can be as expensive as adding all entries in turn sequentially. Our solution is to use a parallel fold function ρ, which, in the applications in this thesis, are just the parallel *join*-based set algorithms (*union*, *intersection* and *difference*) in Chapter 3. We will show the cost bound and analysis for the sweepline algorithms in Section 9.2.3.

Chapter 5

Making Trees Persistent

In this section, we will introduce how to make the *join*-based algorithms persistent, i.e., the algorithms preserve the input tree and create a new version on update. Keeping history versions of an update algorithm is helpful in many applications. For example, in database management systems (DBMSs), it is likely useful to access a previous version of the database for enable rolling backs, or doing analytical queries on an early version. It is especially useful for modern DBMSs on multi-core systems, where concurrent updates and queries need to work consistently on the same database. In this case, preserving the previous version of the DBMS with isolation is essential for the queries to process without being affected or blocked by updates. This technique is called *multi-version concurrency control (MVCC)*. Given this advantage of multiversioning, almost all modern database systems, including Oracle, Microsoft SQL [123, 185], PostgresSQL [233], SAP HANA [133, 251], and HyPer [177, 178], support MVCC. Another example is the implementation of the prefix structures in this thesis. A common scenario of using prefix structures is when the prefix structures are trees, and each update function is an insertion of the next data point. To obtain the whole list of prefix structures, the insertion cannot be in-place, but instead need to preserve both the input and output tree as two versions.

This thesis makes P-Trees persistent by using path-copying, which means to copying all affected nodes on the update path. A simple example of an insertion using path-copying is shown in Figure 1.1(b). All nodes on the insertion path are copied to avoid in-place modification of existing tree nodes. The output is finally represented by the new root pointer, and the input is preserved and can be represented still by the old root pointer. In total $O(\log n)$ tree nodes are copied.

As a result of path-copying, multiple versions of trees share physical tree nodes. This reduces time and space complexity to maintain multiple versions. Such node sharing, however, requires a carefully designed garbage collector to avoid deleting visible nodes or retaining unreachable nodes indefinitely. For P-Trees we implement a reference counting garbage collector (GC) [114, 163, 171]. Each tree node will maintain a *reference counter*

(RC), which records the number of references (pointers) to it, e.g., child pointers from parent nodes, handles to a root, etc.

In the following sections, we present the basic methodology of path-copying especially in *join*-based algorithms. We then introduce how to garbage collect out-of-date tree nodes in old versions. Finally, we discuss and compare path-copying-based approach with other methodologies. In Chapter 11, we will show how to apply persistent P-Trees to an HTAP database system to support MVCC and SI. In Chapter 12, we will present algorithms for version maintenance and memory reclamation on top of the functional data structures.

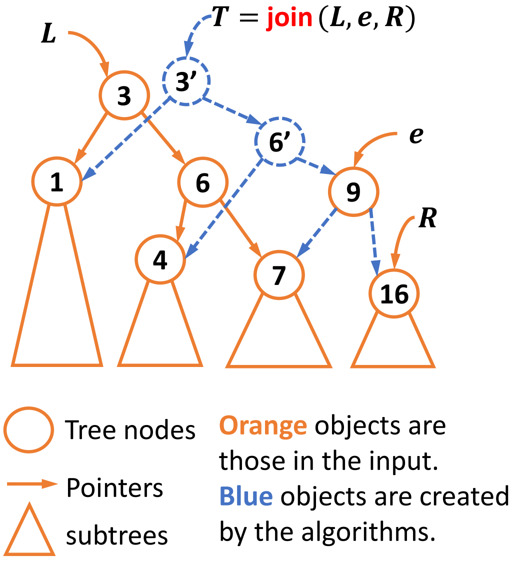
5.1 Purely Functional P-Trees Using Path-copying

Motivation and Related Work. As mentioned, allowing multi-versioning (or persistence) is useful in many applications, but one challenge is to avoid the expensive cost (both in time and in space) of copying the large fraction (if not all) of data of the old version. There are various approaches studied in the previous work. One commonly-used one is to use version chains. which maintain a “history” of versions of each single object as a timestamped linked list, and have each transaction traverse the version list (also called version chains) at every object of the database to find the right version [61, 181, 226, 237]. This avoids copying the whole database when creating new versions. Read-only transactions are also not blocked by writers, and thus the DBMS does not need locks for read transactions.

A drawback of version chains, however, is that finding an entry that is visibile to a transaction requires following pointers and checking the visibility of each individual tuple version. This means that transactions may spend a significant amount of effort traversing these lists when there are a large number of history versions, even if the database itself is small. It also complicates garbage collection since information about a particular version can be spread throughout the version lists. One can reduce this overhead by maintaining meta-data about tuples. For example, the HyPer database creates “version synopses” that identifies whether all of the tuples within a range/block are currently the latest version [214]. This meta-data incurs storage overhead, which is problematic for in-memory systems. Also, although helpful in general, this technique does not provide any worst-case guarantee of acceleration, and might still require to scan the whole list in some cases.

To enable persistence on P-Trees, this thesis adopts path-copying. Instead of updating in place, path-copying always copy the tree node when it is updated, either for an updated entry or a child pointer. In other words, this makes the P-Trees purely functional [45, 173, 219, 232],[1] Such path-copying is the standard approach in functional languages

[1] This is the style used in all functional languages, including Haskell, OCaml, ML, and F#, and is often also used in imperative languages for safety or to maintain persistent copies of data.


T = join (L, e, R)
L
3
3'
6'
e
1
6
9
R
4
7
16
Tree nodes
Pointers
subtrees
Orange objects are those in the input.
Blue objects are created by the algorithms.

```
node* link(node* L, node* e, node* R){
  e->lc = L, e->rc = R, update(e), return e}

/* left_join is symmetric */
node* right_join(node* L, node* e, node* R) {
  if (balance(L, R)) return link(L, e, R)
  node* t = copy(L)
  add_ref(L->lc); add_ref(L->rc); dec_ref(L)
  t->rc = right_concat(t->rc, e, R)
  //rebalance if needed
  return t }

node* join(node* L, node* e, node* R) {
   if (heavier(L, R)) return right_join(L, e, R)
   if (heavier(R, L)) return left_join(R, e, L)
   return link(L, e, R) }
```

node: the tree node type
  lc: left child pointer
  rc: right child pointer

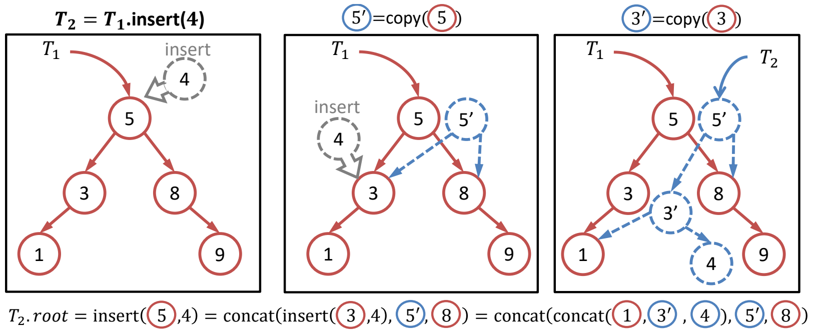
add_ref / **dec_ref** increase / decrease the reference counter by 1.
copy(t): create a new node with the entry in t.
balance and **heavier** depends on the specific balancing scheme.

Figure 5.1: The Functional *join* Algorithm – The algorithm copies all tree nodes on the *join* path.

since the early days of Lisp (1960s), and is also employed in maintaining some previous multi-version data structures [43, 255] and disk-based database systems [4, 26, 64].

***The Functional* join.** The correctness of these algorithms requires the *join* function, as an algorithmic subroutine, to be functional. A functional *join* keeps both versions before and after the operation occurs. In the pseudocode shown in Figure 5.1, we abstract out the common ideas that are independent of balancing schemes to show details about path-copying. Generally, the algorithm goes along the right spine of T_L and copies all the visited nodes, until it finds a node in T_L that is balanced with T_R. Finally, the algorithm *join*s them back using all the copied nodes as pivots. When necessary, the algorithm applies rotations to rebalance the tree, which might also copy a constant number of nodes per rotation. The rebalancing is the same as introduced in Section 3.2. Path-copying does not increase the asymptotical cost of the *join* algorithms. The space overhead, which is the number of copied tree nodes, is of the same order of the time complexity.

With the functional *join*, we can implement all other *join*-based tree algorithms. In addition to using a functional *join* algorithm, the functional counterparts of all other algorithms in Section 3.3 just copy the pivot of *join* before calling *join*. We next show one simple example.



```
node* insert($t$, $\langle k, v\rangle$, $\sigma$) {
  if (t=NULL) return new_node(k, v)
  if (k < t->key){
    node* l ← insert(t->lc, $\langle k, v\rangle$, $\sigma$)
    add_ref(t->rc)
    return concat(l, copy(t), t->rc) }
  if (t->key < k) { /* symmetric */ }
  if (t->key = k) {
    node* t' ← copy(t)
    t'->val ← $\sigma$(t->val, v)
    add_ref(t->lc), add_ref(t->rc)
    return concat(t->lc, t', t->rc) }
}
```

Figure 5.2: The Functional Insertion Using Path-copying – Arrows represent pointers. The function copy(t) create a new node with the same key and value as in t. At the end, the result tree T is represented as the root pointer to $5'$. We note that the algorithm is not specific to any balancing schemes, but only relies on the *join* function to deal with rebalance issues.

Other* join*-based Persistent Algorithms. We use an persistent insertion algorithm as an example to show how to make join-based algorithms functional using path-copying. The *insert*$(t, \langle k, v\rangle, \sigma)$ function is defined as in Section 3.3.

We present an illustration along with the pseudocode in Figure 5.2. As the non-persistent version, this algorithm compares k with the key at the root and recursively inserts $\langle k, v\rangle$ into the left or right subtree. It then copies the current node, and *join*s the two subtrees back using this new node as the pivot. Similarly, an imbalance may occur at this point, but will be resolved by *join*. Also, because the *join* algorithm is persistent, the old tree is never destroyed or updated, but is preserved as is. All the copied nodes will be on the insertion path. There are two base cases. The first one is when t is empty, and the algorithm creates a new node with the entry $\langle k, v\rangle$. The second one is when $k = t$.key, indicating that k is already in the tree. The algorithm then copies t to a new node t', but updates the value to $\sigma(t.\text{val}, v)$. Finally, the copied root node $\boxed{5'}$ will represent the new tree T_2, and the old root still correctly represents T_1. The asymptotical cost of *insert* is not increased because of path-copying. The number of copied tree nodes is also $O(\log n)$.

Using similar approach, we can make all other *join*-based algorithms persistent without affecting the asymptotical work and span.

5.2 Garbage Collection

In this section, we show how to efficiently collect out-of-date tree nodes on P-Tree. In fact, similar approach is applicable to any functional DAG-based data structures accessible via a single entry node. Our approach is based on reference counting, where each tree nodes maintains the number of references to itself. Intuitively, each node's RC is its in-degree in the memory DAG.

We first define the desired properties of GC on P-Trees. When garbage collecting a tree, we want to reclaim all the tree nodes in this tree that is only in this tree, but not to collect any tree nodes that are shared with other trees. For this purpose, we use a *collect*(t) algorithm on a tree node t to collect any tree nodes in t's subtree that is not shared with other trees. For any trees represented by a root pointer, we call them *versions*, and call their roots *version pointers*. We say that a tree node x *belongs to* a version pointer v if x is reachable from v. The correctness of a *collect* algorithm is defined in Definition 11.

Definition 11. *Let x be a tree node, and t be any time during an execution. A* collect *is* correct *if the following conditions hold.*

- *If for each version v that x belongs to,* collect*(v) has terminated by time t, then x has been freed by t.*
- *If there exists a version v that x belongs to for which* collect*(v) has not been called by time t, then x has not been freed by t.*

The* collect *Algorithm. We now present a *collect* algorithm and show its correctness and efficiency. Path-copying causes subsets of the nodes to be shared among versions. To collect the correct tuples, we use *reference counting* (RC) [114, 171]. Each object maintains a count of references to it, and when it reaches 0, it is safe to collect. Since the memory graph is acyclic (forests). This means that RC allows collecting everything [171]. Accordingly, a x =*new_node*(l, e, r) operation creating a node x with entry e increments the reference counters of its children l and r. A newly-created mode x has counter 0. This gets incremented if x becomes the child of another node. Later, when a *collect* is invoked on a tree node x, it first decrements the count of x. Only if the count of x has reached zero, x gets freed, and all children of x are collected recursively. If x's counter is more than one, the *collect* algorithm terminates since x will not be freed, and thus the counts of its descendants will not be decreased then.

Pseudocode of the *collect*() algorithm is given in Algorithm Figure 5.3. We use $x.ref$ to read the current reference counter of node x. *add_ref*(x) and *dec_ref*(x) means to increase and decrease the reference counter of node x. We leave this general on purpose. The simplest way of implementing the counters is via a fetch-and-add object. However, we note that this could introduce unnecessary contention. To mitigate that effect, other options, like dynamic non-zero indicators [9], can be used. Our implementation simply uses an atomic fetch-and-add to update RCs.

We will then show that the *collect* algorithm is correct and fast. In particular, we prove Theorem 5.2.1.

Theorem 5.2.1. *Our* collect *algorithm (Algorithm 5.3) is correct and takes $O(S + 1)$ time where S is the number of freed nodes.*

First we prove that it satisfies the first part of Definition 11.

```
node(l,e,r) {
  Node y = alloc(Node);
  y.entry=e;
  y.ref=0;
  add_ref(l);
  add_ref(r);}}
```

```
void collect(var x) {
  if (x is null) return;
  int c = dec_ref(x);
  if (c ≤ 1) {
    node* l = lc(x); node* r = rc(x);
    free(x);
    collect(l); collect(r);   }}
```

Figure 5.3: The *new_node* and *collect* algorithms.

Lemma 5.2.2. *Let u be a shared tree node. For any shared node w, let V_w be the set of versions that w belongs to. If a* collect *operation has terminated for each version in V_u, then u has been freed.*

Proof. Fix an execution history and a configuration C. Consider the set G of all shared nodes w such that for each version $v \in V_w$, a *collect*(v) operation has terminated. It suffices to show that for each node in G, there is a *collect* operation that frees the tuple and terminates before C.

Notice that G forms a DAG. Furthermore, for each node $w \in G$, G contains every shared node that points to w. This is because a node belongs to all of the versions that its parent belongs to. Therefore we can proceed by structural induction on G.

For the base of the induction, we prove that each of the roots in G that are not pointed by any other tree nodes has been freed by a completed *collect* operation. Let u be some root in G. We just need to show that each increment of u's reference count has a completed *collect*(u) operation corresponding to it. Since u is not pointed by other nodes, all of its references are from the outside handles to the tree. Since each version v that u belongs to has been applied a collect algorithm, the last completed *collect*(u) operation decrement the reference count of u to 0 and frees u.

Now we prove the inductive step by fixing some node u in G and assuming that all of its parents have been freed by some completed *collect* operation. Similar to the base case, we show that each increment of u's reference count has a completed *collect*(u) operation corresponding to it. So we just need to show that for each shared node w that point to u, there is a completed *collect*(u). By the inductive hypothesis, there is a completed *collect* operation that frees w, and we can see from the code that this operation executes a *collect* on u. Therefore one of the completed *collect*(u) operation sets the reference count of u to 0 and frees u. By structural induction, each tuple in G has been freed and this completes the proof. □

Next we prove that our collect algorithm satisfies the second part of Definition 11.

Lemma 5.2.3. *Let u be a shared node and let V_u be the set of versions that it belongs to. If a* collect *operation has not started for some version $v \in V_u$, then u has not been freed.*

Proof. Next we claim that each *collect*(u) operation corresponds to an unique increment of u's reference counter. This can be seen by a close inspection of the code; let c be a *collect*(u) call and consider two cases. Case (1): c is not called from inside another *collect*. That is, u is the root of a version that is being collected. In that case, c corresponds to the increment of $u.ref$ that creates this version. Case (2): c is called recursively from a *collect*(u') algorithm. In this case, the c corresponds to the increment of $u.ref$ during the creation of u'.

Let $v \in V_u$ be the version for which no *collect*(v) call has been invoked. Since u belongs to v, there must be a path from v's version root r to u in the memory graph. We show by induction that no node along that graph has been freed, thus implying that u has not been freed.

Base: Consider v's root, r. $r.ref$ has been incremented when it is created and the *collect*(v) operation corresponding to this increment has not been invoked yet. Therefore the reference count of r is non-zero, so it has not been freed.

Step: Assume that the ith node, u_i in the path from r to u is not freed. We want to show that the $i + 1$th node on this path, u_{i+1} has not been freed either. Consider the *new_node*operation that made u_i the parent of u_{i+1} in the memory graph. That operation incremented u_{i+1}'s reference count by 1 and the *collect*(v) operation corresponding to this increment has not been invoked yet because u_i's RC has not reached 0. Thus, u_{i+1}'s reference count is greater than 0, and therefore it cannot have been freed. □

Finally, we prove that our *collect* algorithm is efficient.

Lemma 5.2.4. *A* collect *operation takes $O(S + 1)$ time where S is the number of nodes that were freed by the operation.*

Proof. Not counting the recursive calls, each *collect* operation needs a constant time. Each time a node is freed, a *collect* operation is called on its two children. Therefore, the total number of *collect* operations spawned by a *collect* operation C is $2S$, where S is the number of nodes that were freed by C. Therefore C has $O(S + 1)$ time complexity in total. □

Together, Lemmas 5.2.2, 5.2.3 and 5.2.4 imply Theorem 5.2.1.

Part II

Implementation and Applications

This part presents several user applications of P-Trees. As a fundamental data structure to maintain ordered data, balanced binary trees themselves are used widely in applications in various areas. Supporting parallelism further improve trees' performance and efficiency. In particular, the symmetric order of balanced binary trees are efficient and elegant for building indexes, supporting fast dynamic updates and lookup. The augmentation on trees provide convenient interface for range-based queries, both for a simple tree structure, and also for multi-dimensional geometry queries. Path-copying on trees enable multi-version concurrency control both correctly and efficiently.

This part will start with the PAM library, including the interface and some implementation details. As the main component, this part will then present applications using P-Trees, include general library interface with range-sums, 1D stabbing queries, 2D range queries, 2D segment queries, 2D rectangle queries, an HTAP database system, and memory reclamation for multi-versioned transactional systems. Along with the applications, we present new algorithms and technologies specific to each application, as well as experimental results.

Experimental Setting

All applications are implemented in C++, based on the PAM library. We use the scheduler in the PBBS library [2, 72]. In many of the implementations, we use the aggregation and sorting algorithms in PBBS.

For all the experiments in this part, 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. In the semantics, we only use the binary fork-join, and parallel loop. We compile with `-O2` because this gives us more stable results. We use `numactl -i all` in all experiments with more than one thread. It evenly spreads the memory pages across the processors in a round-robin fashion. In all applications, we use weight-balanced trees as the balancing scheme because it maintains less metadata in tree nodes (the size is always maintained for all the other balancing schemes), and thus is slightly more space-efficient than other data structures.

Chapter 6

The PAM Library

In this section, we introduce the PAM library, which includes the implementation details and the user interface.

PAM (Parallel Augmented Maps) is a parallel C++ library implementing the interface for augmented maps [262]. It is designed for maintaining multiple ordered structures, including sequences, ordered sets, ordered maps and augmented maps. The library is available on GitHub [261]. The released code includes both the code for the library and the code implementing the applications. It is also designed so it is easy to try in many other scenarios (different sizes, different numbers of cores, etc.).

6.1 ADT Interface

This thesis studies trees that support efficient algorithms for a variety of functions, which, step by step, form interfaces for four abstract data types (ADTs): sequences, ordered sets, ordered maps and augmented maps. Especially, augmented map [262] is a new abstract data type proposed in this thesis, as we introduced in Section 4.1. In this section, we formally define the interface of these ADTs. This is also the interface supported my the PAM library.

All the four data types introduced in this chapter are aggregations of a collection of *entries*, denoted as $\{e_1, e_2, \ldots, e_n\}$. 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 all the four data types. Inside 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 $\subset$, $\in$, $\cup$, $\cap$, and $|\cdot|$ from sets for all the four data types involved.

This thesis will especially present implementations of the four ADTs using balanced binary trees, and show parallel algorithms supporting functions on them in Chapter 3. In the rest of this chapter, we will first demonstrate a high-level overview and motivation of these ADTs, especially for augmented maps. Then in Section 6.1.1–6.1.3, we briefly overview the existing ADTs including sequences, ordered sets and ordered maps, and propose a relatively complete interface for them, respectively. Then we present the augmented maps in details in Section 6.1.4, including the motivation, the formal definition, and the interface.

6.1.1 The Sequence $\mathbb{SQ}(K)$

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[1]. The position of an entry in a sequence is its *index* (or its subscript), and each entry is uniquely retrieved by the 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. $s[i]$ means to extract the i-th element in a sequence s. In this case, no ordering is specified on these entries, and duplicates are allowed. Using a binary tree to maintain such a sequence would just require to organize all entries using the symmetric order (i.e., the in-order traversal) of the tree.

Typically, a sequence $s = \mathbb{SQ}(K)$ supports functions as listed in Table 6.1. Some of the notations are defined in Chapter 2. In the table n denotes the size of s (i.e., $|s|$). In sequences, each key is an entry, and the entry type E is the key type K. The functions on the sequence interface, along with the descriptions are presented in Table 6.1.

6.1.2 Ordered Sets $\mathbb{OS}(K, <_K)$

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, which also disallows duplicates. For binary tree structures, this requires using the symmetric order to maintain the ordering on K, which makes the tree a *binary search tree* (BST) from this point.

In ordered sets, the entry type E of a set is still the same as the key type. The *index* of an entry e in an ordered set st is one plus the number of entries in st that is smaller than (defined by $<_K$) e. The indexes start with 1. Therefore, we denote an ordered set $st = \mathbb{OS}(K)$ of size n as $\{st[1], st[2], \ldots, st[n]\}$ $(st[i] \in K)$ ordered by $<_K$. $st[i]$ means to extract the i-th element in an ordered set st. Beyond the sequence functions, many functions related to the ranking of the entries become useful, such as *next*, *previous*, *upto*, *downto*, etc. Also, many aggregate functions for sets (e.g., *union*, *intersection*, *difference*, etc.) are useful for the interface. Also, all these functions will need to maintain the ordering on the elements.

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

Function		**Description**
empty	:	$\emptyset$, or $\{\}$
domain(s)	:	$\{k(e) \,:\, e \in s\}$
size(s)	:	$\lvert s\rvert$
singleton(e)	:	$\{e\}$
seq_from_array(l)	 :	**Argument:** array $l = [e_1, e_2, \ldots, e_n]$, $e_i \in E$ sequence s, where $s[i] = e_i$
to_array(s)	:	array $[s[1], s[2], \ldots, s[n]]$ 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** $\square$
ind_range(s, i, j)	:	$\{s[i], \ldots, s[j]\}$
(noted as $s[i..j]$**)**	:	
ind_upto(s, i)	:	$\{s[1..i]\}$
ind_downto(s, i)	:	$\{s[i..n]\}$
delete_at(s, i)	:	$\{s_[1..i-1], s_[i+1..n]\}$
insert_at(s, i, e)	:	$\{s[1..i-1], e, s[i..n]\}$
join2(s_1, s_2)	:	$\{s_1[1..\lvert s_1\rvert], s_2[1..\lvert s_2\rvert]\}$
split_at(s, i)	:	$\langle\{s[1..i-1], s[i], s[i+1..\lvert s\rvert]\}\rangle$
filter (s, ψ)	 :	**Argument** $\psi : E \to Bool$ $\{e \in s \mid \psi(e)\}$
map_reduce $(s, f', g', b_\emptyset)$	 :	**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 6.1: 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.

6.1.3 The Ordered Map $\mathbb{OM}(K, <_K, V)$

On top of the ordered set interface, if each key is also assigned a value of type V, then an interface for the ordered *map* can be built, with entries being key-value pairs. The general map structure, also known as key-value store, dictionary, table, or associative array, means to maintain a mapping from keys to values. Accordingly, the *find* function on map m and key k (usually referred to as $m(k)$) returns the value of k in m. As an abstract data type, the map is very important in practice as is indicated by many large-scale data analysis systems such as F1 [250], Flurry [25], RocksDB [239], Oracle NoSQL [221], and LevelDB [191]. Ordered maps means to additionally maintain the ordering on keys, such that range queries and index retrievals are made efficient. To implement an ordered map using binary trees, we only need to store the values as additional information on top of the implementation of ordered sets.

All functions on sequences and sets, as shown in Table 6.1 and 6.2, can be extended for accepting key-value pairs as entries for ordered maps. Based on ordered sets, ordered

Function		Description
from_array(l)	 :	**Argument** $l = [e_1, \ldots, e_n]$ is an array, $e_i \in E$ $\{e \,\vert\, e \in l\}$ ordered by $<_K$ on K
find(st, k)	:	$k \in st$
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)	:	$\{e : e \in st \,\vert\, k(e) \leq k\}$
downto(st, k)	:	$\{e : e \in st \,\vert\, k(e) \geq k\}$
position(st, k)	:	$1 + \lvert\{e \in st \mid k(e) < k\}\rvert$
select(st, i)	:	$st[i]$
split(st, k)	:	$\langle\{e \in st \,\vert\, k(e) < k\}, find(m, k), \{e \in m \,\vert\, k < k(e)\}\rangle$
range(st, k_1, k_2)	:	$\{e \in st \,\vert\, k_1 \leq k(e) \leq k_2\}$
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)	:	$difference(st_1, st_2) \cup difference\ (st_2, st_1)$ $\cup\ intersection\ (st_1, st_2)$

Table 6.2: 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.

maps introduce the value in the entry, so many of the set functions (e.g., *union*, *intersection*, *insert* and *from_array*) 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 6.3. 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 *intersection*, the key will be kept as is, but their values will be combined by σ to be the new value of the key in the output map. 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 *intersection* sum the counts, *insert* add the counts, and *from_array* keep the total counts of the same key in the array. This also applies to when we only need to keep the maximum value of each key among all appearances.

6.1.4 The Augmented Map $\mathbb{AM}(K, <_K, V, aug = \langle A, f, g, I\rangle)$

To equip maps with more functionalities, this thesis proposes the *augmented map* $\mathbb{AM}(K, <_K, V, aug = \langle A, f, g, I\rangle)$ [262], as defined in Definition 9 in Section 4.4. It is an

Function	:	**Description**
find(m, k) **(noted as** $m(k)$**)**	:	$v(e)$ **if** $\exists e \in m, k(e) = k$ **else** $\square$
from_array(l, σ)	:	**Argument** $l = [e_1, \dots, e_n], e_i \in E$ is an array $\{(k', v_{k'}) \mid \forall k' \in domain(l), v_{k'} = \sigma(v_{i_1}, v_{i_2}, \dots v_{i_{n'}}) :$ $\forall v_{i_j}, (k', v_{i_j}) \in s, i_1 < i_2 < \cdots < i_{n'}\}$, ordered by $<_K$ on K
insert (m, e, σ)	:	**Argument** $\sigma : V \times V \to V$ $delete(m, k(e)) \cup \{(k(e), \sigma(v(e), v(Find(m, k(e))))\}$
intersection (m_1, m_2, σ)	:	**Argument** $\sigma : V \times V \to V$ $\{(k, \sigma(v(m_1(k)), v(m_2(k))))\| \, k \in domain(m_1) \cap domain(m_2)\}$
union (m_1, m_2, σ)	:	**Argument** $\sigma : V \times V \to V$ $difference(m_1, m_2) \cup difference\ (m_2, m_1) \cup intersection\ (m_1, m_2, \sigma)$

Table 6.3: 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.

ordered map $\mathbb{OM}(K, <_K, V)$ associated with an augmenting structure. 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 Tables 6.1 to 6.3 should be supported by augmented map interface.

Function	**Description**	**Condition**
aug_val(m)	: $\mathcal{A}(m) = map\_reduce(m, g, f, a_\emptyset)$	-
aug_left(m, k)	: $aug\_val(upto(m, k))$	-
aug_right(m, k)	: $aug\_val(downto(m, k))$	-
aug_range(m, k_1, k_2)	: $aug\_val(range(m, k_1, k_2))$	-
aug_filter(m, ψ)	**Argument** $\psi : A \mapsto Bool$: $Filter(m, \psi')$, where $\psi' : E \mapsto Bool, \psi'(e) \Leftrightarrow \psi(g(e))$	$\forall a, b \in A$, $\psi(a) \vee \psi(b) \Leftrightarrow \psi(f(a, b))$
aug_project (m, g', f', k_1, k_2)	**Argument** $g' : A \mapsto B, f' : B \times B \mapsto B$: $g'(aug\_range(m, k_1, k_2))$	$(B, f', g'(a_\emptyset))$ is a monoid. $f'(g'(a), g'(b)) = g'(f(a, b))$

Table 6.4: The 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.

Table 6.4 presents functions specific to augmented maps along with their descriptions and their applicable conditions. All of them can be defined and 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 the underlying data structure, as in the augmented trees and the prefix structures, as we have shown in Chapter 4. These functions also benefit from using the augmenting functions f and g as operands for plain ordered map functions.

The function *aug_val*(m) returns $\mathcal{A}(m)$, which is equivalent to *map_reduce*$(m, g, f, a_\emptyset)$ 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 *aug_range*(m, k_1, k_2) is equivalent to *aug_val*(*range*(m, k_1, k_2)). Similarly,*aug_left*(m, k) and *aug_right*(m, k) are equivalent to *aug_val*(*upto*(m, k)), respectively. These functions can be implemented using augmented trees with the algorithms in Section 4.2. The *aug_left* algorithm can be implemented using prefix structures by a simple binary search. If an inverse function of f is provided, *aug_right* and *aug_range* can also be implemented in $O(\log n + W_{f^{-1}})$ time, where $W_{f^{-1}}$ denotes the cost of an application of f's inversion function.

Some functions in Table 6.4 are used to accelerate 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 *AugFilter*(m, ψ) function and the *aug_project* (m, g', f', k_1, k_2) are the same as defined in Section 4.2 on augmented trees. Using prefix structures, however, will not accelerate these functions with worst-case guarantee.

6.2 Implementation Details

Parallelism. The core aspect of the PAM library is its use of fork-join parallelism with a dynamic scheduler. The library support both a dynamic scheduler in the PBBS library [72] based on OpenMP, and the Cilk plus scheduler. We also use the parallel aggregation and sorting algorithms from the PBBS library. The parallelism of PAM mainly comes from the divide-and-conquer scheme over the tree structure. As we will show in Section 11.6, PAM achieve almost linear scalability.

To control the granularity of the parallel tasks, the P-Tree only executes two recursive calls in parallel when the current tree size is larger than some threshold. Otherwise the tree processes the operations sequentially. This is to avoid the overhead of forking and joining small tasks. This granularity level is adjustable and is decided by the workload of the base case dealing with a single tree node. Intuitively, if the base cases are light-loaded, we make it more coarse-grained. For example, by default we stop generating parallel tasks when the tree size is under 100. Otherwise, when the work within a single node is sufficient (e.g., they deal with inner trees), we can set the threshold as 1 such that parallelism can be introduced even to the bottom level of the tree.

Memory Management and GC. The P-Tree's memory manager maintains separate pools of nodes for each thread, along with a shared lock-free stack-based pool. The tree maintains blocks of 64k nodes in the shared pool. Each thread retrieves a block from the shared pool when they run out of nodes in their local pool, and then returns a block when they accumulate 2×64k free tree nodes.

We implement a reference counting garbage collector (GC) [163] as introduced in Section 5.2. The system uses an atomic fetch-and-add to update RCs since multiple threads can potentially update an RC concurrently. We use the standard reuse optimization where if a node has a reference count of one (only the current call has a pointer to it), then it is reused rather than being copied [163]. This allows in-place updates when possible.

Durability. All applications introduced in this thesis focus on in-memory systems. However, because P-Trees are functional, each single version can be operated and viewed in isolation. In particular, the root pointers provide snapshots of any history versions of the tree. To make a system using P-Trees durable, one can output such snapshots to disk to preserve the current state. This approach can also be combined with log-based approaches to leverage the storage size and re-construction efficiency.

6.3 User Interface and Artifact Testing

To use the library and define an augmented map using PAM, users need to include the header file `pam.h`, and specify the parameters including type names and (static) functions in an entry structure `entry_type`.

- **typename** `K`: the key type (K),
- **function** `comp`: $K \times K \mapsto$ **bool**: the comparison function on K ($<_K$)
- **typename** `V`: the value type (V),
- **typename** `A`: the augmented value type (A),
- **function** `base`: $K \times V \mapsto A$: the base function (g)
- **function** `combine`: $A \times A \mapsto A$: the combine function (f)
- **function** `identity`: $\emptyset \mapsto A$: the identity of f (I)

This is an entry for an augmented maps. Then an augmented map is defined with C++ template as `aug_map<entry_type>`. For sequences, ordered sets, ordered maps, fewer arguments are required. In particular, a sequence `pam_seq<entry_type>` only needs the key type `K` defined in the `entry`. An ordered set (`pam_set<entry_type>`) only needs the key type `K` with the `comp` function. An ordered map (`pam_map<entry>`) only requires `K`, `comp` and `V`.

Here is an example of defining an augmented map m that has integer keys and values and is augmented with value sums:

```
struct entry {
  using key_t = int;
  using val_t = int;
  using aug_t = int;
  static bool comp(key_t a, key_t b) {return a < b;}
  static aug_t I() { return 0;}
  static aug_t base(key_t k, val_t v) {return v;}
  static aug_t combine(aug_t a, aug_t b) {return a+b;}};
aug_map<entry> m;
```

Another quick example can be found in Chapter 8, which shows how to implement an interval tree using the PAM interface.

The library also provides an implementation of prefix structures in `common/` directory. In particular, one can construct prefix structures using:

```
A* sweep(E* data, size_t n, A Init, F f, G g, H h, size_t num_blocks=0)
```

The arguments are specified as follows.

- **array head pointer** `data`: the head pointer to an array storing all the input entries,
- **integer** `n`: the length of `A`,
- **augmented value type** `Init`: the identity of the augmented value regarding combine function f,
- **function** `f`: the combine function,
- **function** `g`: the fold function,
- **function** `h`: the update function,
- **integer** `num_blocks`: the number of blocks used by the parallel construction algorithm Algorithm 1. When this value is set to 0, the algorithm automatically sets it as the number of physical threads available in the machine.

This function will return an array of augmented values, which are the prefix structures in order.

Library meta information.

- **Algorithm:** Join-based balanced binary tree algorithms, the prefix structures, and example applications of them, including range sum, interval trees, 2D range/segment/rectangle queries, inverted index searching, an HTAP database system for TPC-H queries and TPC-C-like transactions.
- **Program:** C++ code supporting both the Cilk Plus extensions and OpenMP for parallelism.
- **Compilation:** g++ 5.4.0 (or later versions).
- **Run-time environment:** Linux. The scripts that we provide in the repository use `numactl` for better performance. All tests can also run directly without `numactl`.
- **Hardware:** Any modern (2010+) x86-based multicore machines. Relies on 128-bit CMPXCHG (requires `-mcx16` compiler flag) but does not need hardware transactional memory (TSX).

- **Experiment workflow:** See instructions in the GitHub repository.

Chapter 7

General Operations and Range Sum Queries

Our first application is a simple ordered map storing 64-bit integers as keys and values, and an augmentation of keeping the sum of all values (or the maximum value) in each subtree. The tree can be represented as given in Equation (4.6).

7.1 Experiments

We test the performance of multiple functions on this structure. We also compare PAM with some sequential and parallel libraries. None of the other implementations support augmentation. We use 64-bit integer keys and values. The results on running time are summarized in Table 7.1. Our times include the cost of any necessary garbage collection (GC). We also present space usage in Table 9.5.

We test versions both with and without augmentation. For general map functions like *union* or *insert*, maintaining the augmented value in each node costs overhead, but it seems to be minimal in running time (within 10%). This is likely because the time is dominated by the number of cache misses, which is hardly affected by maintaining the augmented value. The overhead of space in maintaining the augmented value is 20% in each tree node (extra 8 bytes for the augmented value). For the functions related to the range sum, the augmentation is necessary for theoretical efficiency, and greatly improves the performance. For example, the *aug_range* function using a plain (non-augmented) tree structure would require scanning all entries in the range, so the running time is proportional to the number of related entries. It costs 0.44s to process 10^4 parallel *aug_range* queries. With augmentation, *aug_range* has performance that is close to a simple *find* function, which is only about 3s for 10^8 queries. Another example to show the advantage of augmentation is the *aug_filter* function. Here we use *Max* instead of taking the sum as the combine function, and set the filter function as selecting all entries

	n	**m**	**$\mathbf{T_1}$**	**$\mathbf{T_{144}}$**	**Spd.**
PAM (with augmentation)					
Union	10^8	10^8	12.517	0.2369	52.8
Union	10^8	10^5	0.257	0.0046	55.9
Find	10^8	10^8	113.941	1.1923	95.6
Insert	10^8	–	205.970	–	–
Build	10^8	–	16.089	0.3232	49.8
Build	**$\mathbf{10^{10}}$**	–	**1844.38**	**28.24**	**65.3**
Filter	10^8	–	4.578	0.0804	56.9
Multi-Insert	10^8	10^8	23.797	0.4528	52.6
Multi-Insert	10^8	10^5	0.407	0.0071	57.3
Range	10^8	10^8	44.995	0.8033	56.0
AugLeft	10^8	10^8	106.096	1.2133	87.4
AugRange	10^8	10^8	193.229	2.1966	88.0
AugRange	**$\mathbf{10^{10}}$**	**$\mathbf{10^8}$**	**271.09**	**3.04**	**89.2**
AugFilter	10^8	10^6	0.807	0.0163	49.7
AugFilter	10^8	10^5	0.185	0.0030	61.2
Non-augmented PAM (general map functions)					
Union	10^8	10^8	11.734	0.1967	59.7
Insert	10^8	–	186.649	–	–
build	10^8	–	15.782	0.3008	52.5
Range	10^8	10^8	42.756	0.7603	56.2
Non-augmented PAM (augmented functions)					
AugRange	10^8	10^4	21.642	0.4368	49.5
AugFilter	10^8	10^6	2.695	0.0484	55.7
AugFilter	10^8	10^5	2.598	0.0497	52.3
STL					
Union Tree	10^8	10^8	166.055	–	–
Union Tree	10^8	10^5	82.514	–	–
Union Array	10^8	10^8	1.033	–	–
Union Array	10^8	10^5	0.459	–	–
Insert	10^8	–	158.251	–	–
MCSTL					
Multi-Insert	10^8	10^8	51.71	7.972	6.48
Multi-Insert	10^8	10^5	0.20	0.027	7.36

Table 7.1: Timings in seconds for various functions in PAM, the C++ Standard Template Library (STL) and the library Multi-core STL (MCSTL) [252] – Here “T_{144}” means on all 72 cores with hyperthreads (i.e., 144 threads), and “T_1” means the same algorithm running on one thread. “Spd.” means the speedup (i.e., T_1/T_{144}). For insertion we test the total time of n insertions in turn starting from an empty tree. All other libraries except PAM are not augmented.

with values that are larger than some threshold θ. In this case whenever the algorithm is processing some tree node with augmented value smaller than θ, the whole subtree

Func.	Type	Overhead for aug.			Saving from node-sharing		
		node size	aug. size	over-head	#nodes in theory	Actual #nodes	Saving ratio
Union	m = 10^8	48B	8B	20%	390M	386M	1.2%
	m = 10^5	48B	8B	20%	200M	102M	49.0%

Table 7.2: Space used by the *union* function – We use B for byte, M for million. The overhead for augmentation is computed by aug_size/(node_size-aug_size). The saving ratio from persistence is computed by 1 - #actual/#in_theory.

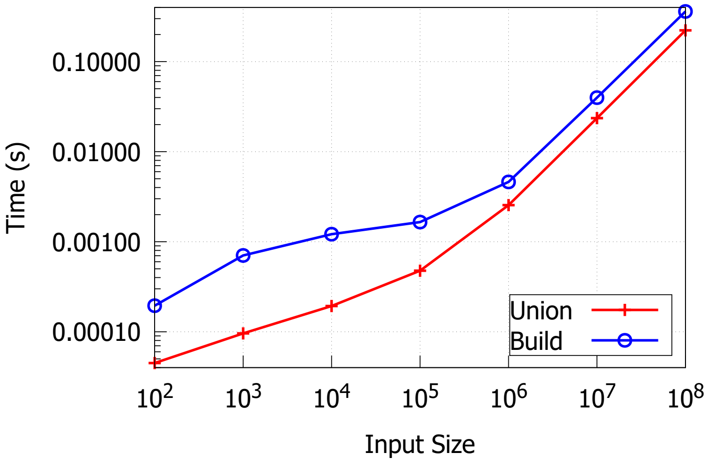


Figure 7.1: The running time of *union* and *build* using PAM on different input sizes – Results collected on a machine with 72 cores with hyperthreading.

can be filtered out. We set the parameter m as the output size, which can be adjusted by choosing appropriate θ. Such an algorithm has theoretical work of $O(m\log(n/m+1))$, and is significantly more efficient than a plain implementation (linear work) when $m \ll n$. We give two examples of tests on $m = 10^5$ and 10^6. The change of output size does not affect the running time of the non-augmented version, which is about 2.6s sequentially and 0.05s in parallel. When making use of the augmentation, we get a 3× improvement when $m = 10^6$ and about 14× improvement when $m = 10^5$.

For sequential performance we compare to the C++ Standard Template Library (STL) [212], which supports `set_union` on sets based on red-black trees and sorted vectors (arrays). We denote the two versions as *Union-Tree* and *Union-Array*. In Union-Tree results are inserted into a new tree, so it is also persistent. When the two sets have the same size, the array implementation is faster because of its flat structure and better cache performance. If one map is much smaller, PAM performs better than Union-Array because of better theoretical bound ($O(m\log(n/m+1))$ vs. $O(n+m)$). It outperforms Union-Tree because it supports persistence more efficiently, i.e., sharing nodes instead of making a copy of all output entries. Also, our *join*-based *insert* achieves performance close to (about 17% slower) the well-optimized STL tree insertion even though PAM needs to maintain the reference counts. We also compare to Intel TBB [166, 231] concurrent hash map, which is a parallel implementation on *unordered* maps. On inserting $n = 10^8$ entries into a pre-allocated table of appropriate size, it takes 0.883s compared to our 0.323s (using all 144 threads).

In parallel, the speedup on the aggregate functions such as *union* and *build* is above 50. The functions that are "embarrassingly parallel" (such as *find*) have up to 90-fold speedup. The range functions *range* and *aug_range* both take less than 0.1 μs per query in parallel. Generally, the speedup is correlated to the ratio of reads to writes. With all (or mostly) reads to the tree structure, the speedup is often more than 72 (number of cores) with hyperthreads (e.g., *find*, *aug_left* and *aug_range*). With mostly writes (e.g., building a new tree as output) it is 40-50 (e.g., *filter*, *range*, *union* and *aug_filter*). The *build* function is relatively special because the parallelism from the parallel sorting is also significant. We also give the performance of the *multi_insert* function in the Multicore STL (MCSTL) [252] for reference. On our server MCSTL does not scale to 144 threads, and we show the best time it has (on 8-16 threads). On the functions we test, PAM outperforms MCSTL both sequentially and in parallel.

PAM is scalable to very large data, and still achieve very good speedup. On our machine, PAM can process up to 10^{10} elements (highlighted in Table 7.1). We give the running time on building a tree of 10^{10} entries and running *aug_range* on it. It takes more than half an hour to build the tree sequentially, but only needs 28 seconds in parallel, achieving a 65-fold speedup. For *aug_range* the speedup is about 90.

Also, using path-copying to implement persistence improves space-efficiency. For the persistent *union* on two maps of size 10^8 and 10^5, we save about 49% of tree nodes because most nodes in the larger tree are re-used in the output tree. When the two trees are of the same size and the keys of both trees are extracted from the similar distribution, there is little savings.

We present the parallel running times of *union* and *build* on different input sizes in Figure 7.1. For *union* we set one tree of size 10^8 and vary the other tree size. When the tree size is small, the parallel running time does not shrink proportional to size (especially for *build*), but is still reasonably small. This seems to be caused by insufficient parallelism on small sizes. When the input size is larger than 10^6, the algorithms scales very well.

Chapter 8

Interval Trees

This chapter gives an example of how to use our interface for interval trees [118, 119, 129, 134, 180, 199]. This data structure maintains a set of intervals on the real line, each defined by a left and right endpoint. Various queries can be answered, such as a stabbing query which given a point reports whether it is in an interval.

8.1 Approach

There are various versions of interval trees. Here we discuss the version as described in [118]. In this version each interval is stored in a tree node, sorted by the left endpoint (key). A point x is covered by in an interval in the tree if the maximum right endpoint for all intervals with keys less than x is greater than x (i.e. an interval starts to the left and finishes to the right of x). By storing at each tree node the maximum right endpoint among all intervals in its subtree, the stabbing query can be answered in $O(\log n)$ time. An example is shown in Figure 8.2.

In our framework this can easily be implemented by using the left endpoints as keys, the right endpoints as values, and using max as the combining function. The definition is:

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

Figure 8.1 shows the C++ code of the interval tree structure using PAM. The entry with augmentation is defined in `entry` starting from line 3, containing the key type `key_t`, value type `val_t`, comparison function `comp`, augmented value type (`aug_t`), the base function g (`base`), the combine function f (`combine`), and the identity of f (`identity`). An augmented map (line 12) is then declared as the interval tree structure with `entry`. The constructor on line 14 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 `amap::aug_left(m,p)`. As defined in Chapter 4, this function returns the augmented sum, which is the max on values, of all entries with keys less than p. As mentioned we need only to compare it with p. The function `report_all(p)` returns all intervals containing p, which are those with keys less than p but values larger than p. We first get the sub-map in `m` with keys less then p (`amap::upTo(m,p)`), and filter all with values larger than p. Note that $h(a) = (a > p)$ and the combine function $f(a, b) = \max(a, b)$ satisfy $h(a) \vee h(b) \Leftrightarrow h(f(a, b))$. This means that to get all nodes with values $> p$, if the maximum value of a subtree is less than p, the whole subtree can be discarded. Thus we can apply `amap::aug_filter` ($O(k \log(n/k + 1))$ work for k results), which is more efficient than a plain filter.

```
struct interval_map {
  using interval = pair<point, point>;
  struct entry {
    using key_t = point;
    using val_t = point;
    using aug_t = point;
    static bool comp(key_t a, key_t b) { return a < b;}
    static aug_t identity() { return 0;}
    static aug_t base(key_t k, val_t v) { return v;}
    static aug_t combine(aug_t a, aug_t b) {return (a > b) ? a : b;}
  };
  using amap = aug_map<entry>;
  amap m;

  interval_map(interval* A, size_t n) {
    m = amap(A,A+n); }

  bool stab(point p) {
    return (amap::aug_left(m,p) > p);}

  amap report_all(point p) {
    amap t = amap::up_to(m,p);
    auto h = [] (P a) -> bool {return a>p;}
    return amap::aug_filter(t,h);}

  amap insert(interval i) {
    amap::insert(m,i,[](point a, point b) {return max(a,b);});
  };
```

Figure 8.1: The definition of interval maps using PAM in C++.

We note that beyond being very concise the interface gives a large amount of functionality, including the ability to take unions and intersections of interval maps, taking ranges, or applying a filter. For example, filtering out all intervals less than a given length `L` can be implemented as:

```
void remove_small(int L) {
  auto f = [&] (interval I) {
   return (I.second - I.first >= L);};
```

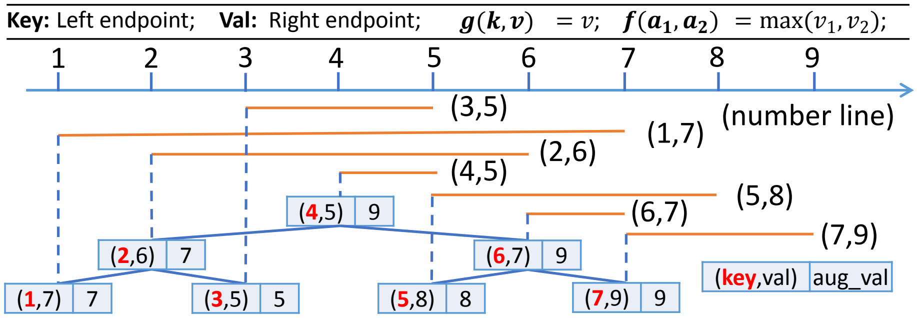


Figure 8.2: An example of an interval tree.

```
  m.filter(f); }
```

It also runs in parallel.

In Section 8.1.1 we report on the performance of the implementation. It is by far the fastest implementation of interval trees we know of. On one core it can build an interval tree on 1 billion intervals in about 180 seconds, and on 72 cores it can build it in about 3.5 seconds. The python implementation [154], by comparison, takes about 200 seconds to build a tree on just 10 million intervals. Although unfair to compare performance of C++ to python (python is optimized for ease of programming and not performance), the code for interval trees based on our library is much simpler than the python code—30 lines in Figure 8.1 vs. over 2000 lines of python. This does not include our code for for the PAM library itself (about 4000 lines of code), but the point is that our code can be shared among many applications while the Python library is specific for the interval query. Also our code has much more functionality, including many more functions, and support for parallelism.

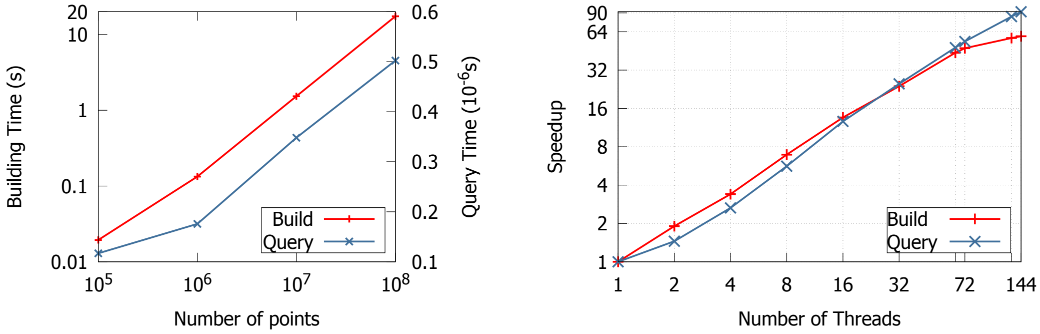
8.1.1 Experiments

We test our interval tree (same code as in Figure 8.1) using the PAM library. For queries we test 10^9 stabbing queries. We give the results of our interval tree on 10^8 intervals in Table 8.1 and the speedup figure in Figure 8.3.

We compare the sequential performance of our interval tree with a Python interval tree implementation [154]. The Python implementation is sequential, and is very inefficient. We ran the Python interval tree only up to a tree size of 10^7 because it already took about 200s to build the tree when the size is 10^7, and can hardly accept larger trees as input. The sequential running time of our code and the Python interval tree is reported in Table 8.1. In building the tree, our code is about dozens of times more efficient than the Python implementation, and in performing queries, is orders of magnitude faster. Our code can answer millions of queries in one second, while the Python interval tree can only do 418 per second when n is as small as 10^4, and this processing rate deceases dramatically when

n	Build		Queries	
	Python Melts/sec	PAM Melts/sec	Python queries/sec	PAM queries/sec
10^4	0.102	8.264	417.981	13.333×10^6
10^5	0.059	17.147	26.105	28.169×10^6
10^6	0.064	16.437	2.520	24.038×10^6
10^7	0.049	13.063	0.209	22.845×10^6
10^8	-	12.247	-	21.067×10^6

Table 8.1: The processing rate of the Python interval tree and PAM interval tree on n points – The "Build" column shows the millions of input entries processed per second (calculated as n/(build-time$\times10^6$)). In "Queries" we show the number of queries processed per second (calculated as query-number/query-time).



(a) (b)

Figure 8.3: The performance of the interval tree using PAM – Figure (a) shows the sequential running time of PAM interval tree on construction (left y-axis) and performing one query (right y-axis) respectively. Figure (b) shows the speedup on various numbers of processors with 10^8 points in the tree.

n gets larger. The sequential running time of our implementation is also given in Figure 8.3(a).

Sequentially, even on 10^8 intervals the tree construction only takes 14 seconds, and each query takes around 0.58 μs. We did not find any comparable open-source interval-tree library in C++ to compare with. The only available library is a Python interval tree implementation [154], which is sequential, and is very inefficient (about 1000 times slower sequentially). Although unfair to compare performance of C++ to python (python is optimized for ease of programming and not performance), our interval tree is much simpler than the python code—30 lines in Figure 8.1 vs. over 2000 lines of python. This does not include our code in PAM (about 4000 lines of code), but the point is that our library can be shared among many applications while the Python library is specific for the interval query. Also our code supports parallelism. We ran the Python interval tree only up to a tree size of 10^7 because it already took about 200s to build the tree when the size is 10^7, and can hardly accept larger trees as input. Our code can answer millions of queries in one second, while the Python interval tree can only do 418 per second when n is as small as 10^4, and this processing rate deceases dramatically when n gets larger. Even considering the language, our implementation should be much efficient than Python library.

In parallel, on 10^8 intervals, our code can build an interval tree in about 0.23 second, achieving a 63-fold speedup. We also give the speedup of our PAM interval tree in Figure 8.3 (b). Both construction and queries scale up to 144 threads (72 cores with hyperthreads). The building process gets almost-linear speedup when less than 72 threads are involved and improves slightly after that. For queries, all queries run independently in parallel, and the speedup is almost perfectly-linear. The speedup of query is over 90 on all 144 threads.

Chapter 9

2D Range-based Geometry Problems

9.1 Introduction

Range, segment and rectangle queries are fundamental problems in computational geometry, with extensive applications in many domains. In this thesis, we focus on 2D Euclidean space. The range query problem is to maintain a set of points, and to answer queries regarding the points contained in a query rectangle. The segment query problem is to maintain a set of non-intersecting segments, and to answer questions regarding all segments intersected with a query vertical line. The rectangle stabbing query (also referred to as the *enclosure stabbing query*) problem is to maintain a set of rectangles, and to answer questions regarding rectangles containing a query point. For all problems, we discuss queries of both listing all queried elements (the *list-all* query), and returning the count of queried elements (the *counting* query). Some other queries, can be implemented by variants (e.g., the weighted sum of all queried elements) or combinations (e.g. rectangle-rectangle intersection queries) of the queries in this thesis. Efficient solutions to these problems are mostly based on variants of range trees [57], segment trees [52], sweepline algorithms [249], or combinations of them.

In addition to the large body of work on sequential algorithms and data structures [56, 58, 101, 106, 130, 131], there have also been many theoretical results on parallel algorithms and structures for such queries [17, 32, 35, 141]. Some of our algorithms are also motivated by previous theory work on efficient parallel algorithms [17, 32]. However, efficient implementations of these structures can be complicated. We know of few parallel implementations of these theoretically efficient query structures, primarily due to delicate design of algorithmic details required by the structures. The parallel implementations we know of [100, 161, 172, 205] do not have useful theoretical bounds. Our goal is to develop theoretically efficient algorithms which can be implemented with ease and also run fast in practice, especially *in parallel*.

This thesis solves these problems using the augmented map framework. We use augmented maps to help develop efficient and concise implementations. To do this, there are two major steps: 1) modeling problems using augmented maps, and 2) implement augmented maps using efficient data structures. Indeed, as shown in Section 9.3, 9.4 and 9.5, we model the range, segment and rectangle query problems all as two-level map structures: an outer level map augmented with an inner map structure. We show that the augment map abstraction is extendable to a wide range of problems, and develop five structures on top of the augmented map interface corresponding to different problems and queries.

As for implementing augmented maps, we employ two data structures: P-Trees with augmentation, and *prefix structures*. Especially, we specify the function parameters of the prefix structures for the interested geometry problems. We then discuss their parallel implementations and analyze cost bounds. Interestingly, the algorithms based on the prefix structures resemble the standard sweepline algorithms. Therefore, our algorithm also parallelizes a family of sweepline algorithms that are efficient both in theory and practice. As a result, both augmented trees and prefix structures provide efficient parallel implementations for augmented maps, and each has its own merits in different settings.

By combining the five two-level map structures with the two underlying data structures as the outer map (the inner maps are always implemented by augmented trees), we develop a total of ten different data structures for range, segment and rectangle queries. Among the ten data structures, five of them are multi-level trees including *RangeTree* (for range query), *SegTree* (for segment query), *RecTree* (for rectangle query), and another two for fast counting queries *SegTree** (segment counting) and *RecTree** (rectangle counting). The other five are the corresponding sweepline algorithms.

All the data structures in this chapter are efficient in theory. We summarize the theoretical costs in Table 1.3. The construction bounds are all optimal in work (lower bounded by sorting), and the query time is almost-linear in the output size. We did not use fractional cascading [103], so some of our query bounds are not optimal. However, we note that they are sub-optimal by at most a $\log n$ factor.

All the data structures in this chapter are also fast in practice. We implement all of them making use of a parallel augmented map library *PAM* [262], which supports augmented maps using augmented trees. We compare our implementations to C++ libraries CGAL [98] and Boost [5]. We achieve a 33-to-68-fold self-speedup in construction on 72 cores (144 hyperthreads), and 60-to-126-fold speedup in queries. Our sequential construction is more than 2x faster than CGAL, and is comparable to Boost. Our query time outperforms both CGAL and Boost by 1.6-1400x. We also provide a thorough comparison among the new algorithms in this chapter, leading to many interesting findings.

Beyond being fast, our implementation is also concise and simple. Using the augmented map abstraction greatly simplifies engineering and reduces the coding effort, which is indicated by the required lines of code—on top of PAM, each application only requires about 100 lines of C++ code even for the parallel version. For the same functionality, both CGAL and Boost use hundreds of lines of code for each sequential implementation. We note that PAM implements general-purpose augmented maps, and does not directly provide anything special for computational geometry.

Notation. In two dimensions, let X, Y and $D = X \times Y$ be the types of x- and y-coordinates and the type of points, where X and Y are two sets with total ordering defined by $<_X$ and $<_Y$ respectively. For a point $p \in D$ in two dimensions, we use $x(p) \in X$ and $y(p) \in Y$ to extract its x- and y-coordinates, and use a pair $(x(p), y(p))$ to denote p. For simplicity, we assume all input coordinates are unique. Duplicates can be resolved by slight variations of algorithms in this chapter.

9.2 Parallel Sweepline Algorithms Using Prefix Structures

9.2.1 Sweepline Algorithms

A sweepline algorithm (or plane sweep algorithm) is an algorithmic paradigm that uses a conceptual sweep line to process elements in order [249]. It uses a virtual line sweeping across the plane, which stops at some points (e.g., the endpoints of segments) to make updates. We refer to the points as the *event points* $p_i \in P$. They are processed in a total order defined by $\prec: P \times P \mapsto \mathit{Bool}$. Here we assume the events are known ahead of time. As the algorithm processes the points, a data structure T is maintained and updated at each event point to track the status at that point. Sarnak and Tarjan [242] first noticed that by being persistent, one can keep the intermediate structures $t_i \in T$ at all event points for later queries. In this chapter, we adopt the same methodology, but parallelize it.

Typically in sweepline algorithms, on encountering an event point p_i we compute t_i from the previous structure t_{i-1} and the new point p_i using an *update function* $h : T \times P \mapsto T$, i.e., $t_i = h(t_{i-1}, p_i)$. The initial structure is t_0. A sweepline algorithm can therefore be defined as the five tuple:

$$\boxed{\mathrm{S} = \mathbf{SW}\ (\ \boldsymbol{P};\quad \boldsymbol{\prec}: P \times P \mapsto \mathit{Bool};\quad \boldsymbol{T};\quad \boldsymbol{t_0} \in T;\quad \boldsymbol{h} : P \times T \mapsto T\ )}$$

It defines a function that takes a set of points $p_i \in P$ and returns a mapping from each point to a data structure $t_i \in T$.

9.2.2 Using Prefix Structures for Sweepline Algorithms

In many non-trivial instantiation of the sweepline algorithms, especially the applications discussed in this chapter, the update function is associative. In this case, the data structures of the sweepline algorithm is equivalent to maintaining the prefix structures of an augmented map. In particular, we store the sequence of event points in order in a map m, and at each point p_i we use the augmented value of the prefix in m up to p_i as the prefix structure. This is equivalent to using a combination of function f and g as the update function. That is to say, an augmented map $m = \mathbf{AM}(K, \prec, V, A, f, g, a_\emptyset)$ can be implemented by a sweepline paradigm S as:

$$S = \mathbf{SW}(K \times V; \prec; A; t_0 \equiv I; h(t,p) \equiv f(t, g(p))) \tag{9.1}$$

In such a sweepline algorithm, we accordingly also call t_i the *prefix structure* at the event point i.

We present a parallel algorithm to build the prefix structures, assuming the update function h can be applied "associatively" to the prefix structures. We assume $h(t,p) \equiv f_h(t, g_h(p))$ for some associative function $f_h : T \times T \mapsto T$ and $g_h : P \mapsto T$. Similarly as in an augmented map, we call f_h and g_h the *base* and *combine* function of the corresponding update function, respectively. Because of the associativity of h_p, to repeatedly update a sequence of points $\langle p_i \rangle$ onto a "prefix sum" t using h is equivalent to combining the "partial sum" of all points $\langle p_i \rangle$ to t using the combine function f_h.

Meanwhile, S trivially fits the parallel sweepline paradigm as defined in Equation 4.9 as follows:

$$S' = \mathbf{PS}(\boldsymbol{K \times V}; \quad \boldsymbol{\prec}; \quad \boldsymbol{A}; \quad \boldsymbol{t_0} \equiv a_\emptyset; \quad \boldsymbol{h(t,p)} \equiv f(t, g(p)); \quad \boldsymbol{\rho_{f,g}}; \quad f) \tag{9.2}$$

where $\boldsymbol{\rho_{f,g}}(p_1, \ldots p_n) \equiv f(g(p_1), \ldots g(p_n))$. This means we can apply Algorithm 1 to construct such an augmented map in parallel.

The prefix structures are especially useful for queries related to *aug_left*, as is the case for many queries in the three applications in this chapter. When using the prefix structures to represent the outer map in range, segment and rectangle queries, the algorithms are equivalent to sweepline algorithms, and they all accord with the assumption on the function cost in Theorem 9.2.1.

9.2.3 Constructing Cost Bounds

As mentioned in Section 4.5, for reasonably complicated augmentations, especially the sweepline algorithms that will be discussed in this chapter, trivially applying Algorithm 1 is not sufficient for guaranteeing good parallelism. This is because applying the combine function b times in the sweeping step can be as expensive as the original sequential sweepline algorithm, which is the reason that similar attempts previously did not achieve useful depth bounds [107, 205] for parallel sweepline algorithms. We observed that in many instantiations of this framework (especially those in this chapter), a parallel combine function can be applied, which effectively guarantees the work-efficiency and parallelism of this algorithm. We formalize the typical setting as follows. In this chapter, we carefully study the properties of the combine function f and its instantiations of the geometry queries. Such properties guarantee the work-efficiency, parallelism and the ease of coding efforts, and will be explained in the next several paragraphs. We formalize the typical setting as follows.

	The update function ($h(t,p)$)	**The fold function ($\rho(s)$)**	**The combine function ($f(t,t')$)**
Work	$\log \lvert t\rvert$	$\lvert s\rvert \log \lvert s\rvert$	$\lvert t'\rvert \log(\frac{\lvert t\rvert}{\lvert t'\rvert} + 1)$
Depth	$\log \lvert t\rvert$	$\log \lvert s\rvert$	$\log \lvert t\rvert \log \lvert t'\rvert$
Output	$\lvert t\rvert$	$\lvert s\rvert$	$\lvert t'\rvert + \lvert t\rvert$

Table 9.1: A typical setting of the function costs in a sweepline paradigm – Bounds are in Big-O notation.

In a common setting of sweepline algorithms, each prefix structure keeps an ordered set tracking some elements related to a subset of the processed event points, having size $O(i)$ at point i. The function h updates one element to the set at a time, which costs $O(\log n)$ on a prefix structure of size n. The corresponding combine function f_h is some set functions (e.g., a *Union* for a sequence of insertions). Using trees to maintain the sets, previous work shows [74] that the set function can be done in $O(n_2 \log(n_1/n_2 + 1))$ work and $O(\log n_1 \log n_2)$ depth for two set of size n_1 and $n_2 \leq n_1$. This algorithm is implemented in the PAM library (as shown in Table 6.3). Creating the partial sum of a block of r points costs $O(r \log r)$ work and $O(\log r)$ depth, building a prefix structure of size at most r. We summarize the setting in Table 9.1, and the corresponding bounds of the sweepline algorithm in Theorem 9.2.1.

Theorem 9.2.1. *A sweepline paradigm S as in Equation 9.2 can be built in parallel using its corresponding parallel paradigm S' (Equation 4.9). If the bounds as shown in Table 9.1 hold, then Algorithm 1 can construct all prefix structures in work $O(n \log n)$ and depth $O(\sqrt{n} \log^{1.5} n)$, where n is the number of event points.*

Proof. The cost of the algorithm is analyzed as follows:

1. **Batching**. This step builds b prefix structures, each of size at most n/b, so it takes work $O(b \cdot \frac{n}{b} \log \frac{n}{b}) = O(n \log \frac{n}{b})$ and depth $O(\log \frac{n}{b})$.
2. **Sweeping**. This step invokes b times of the combine function f_h sequentially, but the combine function works in parallel. The size of t'_i is no more than $O(r)$. Considering the given work and depth bounds of the combine function, the total work of this step is bounded by: $O\left(\sum_{i=1}^{b} r \log(\frac{ir}{r} + 1)\right) = O(n \log b)$. The depth is: $O\left(\sum_{i=1}^{b} \log r \log ir\right) = O(b \log \frac{n}{b} \log n)$.
3. **Refining**. This step computes all the other prefix structures using h. The total work is: $O\left(\sum_{i=1}^{n} \log i\right) = O(n \log n)$. We process each block in parallel, so the depth is $O(\frac{n}{b} \log n)$.

In total, it costs work $O(n \log n)$ and depth $O\left(\left(b \log \frac{n}{b} + \frac{n}{b}\right) \log n\right)$. When $b = \Theta(\sqrt{n/\log n})$, the depth is $O(\sqrt{n} \log^{1.5} n)$. □

This parallel algorithm is also easy to implement. Our code is available online at `https://github.com/cmuparlay/PAM/blob/master/common/sweep.h`. Theoretically, by repeatedly applying this process to each block in the last step, we can further reduce the depth to $O(n^{-\epsilon})$ for any constant $\epsilon > 0$. We state and prove the following corollary.

Corollary 9.2.2. *A sweepline paradigm S as in Equation 9.2 can be parallelized using its corresponding parallel paradigm S' (Equation 4.9). If the bounds as shown in Table 9.1 hold, then we can construct all prefix structures in work $O(\frac{1}{\epsilon}n\log n)$ and depth $\tilde{O}(n^{\epsilon})$ for arbitrary small $\epsilon > 0$.*

Proof. To reduce the depth of the parallel sweepline paradigm, we adopt the same algorithm as introduced in Theorem 9.2.1, but in the last refining step, repeatedly apply the same algorithm on each block. If we repeat for a c of rounds, for the i-th round, the work would be the same as splitting the total list into k^i blocks. Hence the work is still $O(n\log n)$ every round. After c rounds the total work is $O(cn\log n)$.

For depth, notice that the first step costs logarithmic depth, and the second step, after c iterations, in total, requires depth $\tilde{O}(cb)$ depth. The final refining step, as the size of each block is getting smaller and smaller, the cost of each block is at most $O(\frac{n}{b^i}\log n)$ in the i-th iteration. In total, the depth is $\tilde{O}\left(cb+\frac{n}{b^c}\right)$, which, when $b = c^{\frac{c}{c+1}}n^{\frac{1}{c+1}}$, is $\tilde{O}(n^{1/(c+1)})$. Let $\epsilon = 1/(c+1)$, which can be arbitrary small by adjusting the value of c, we can get the bound in Corollary 9.2.2. □

Specially, when $c = \log n$, the depth will be reduced to polylogarithmic, and the total work is accordingly $O(n\log^2 n)$. This is equivalent to applying a recursive algorithm (similar to the divide-and-conquer algorithm of the prefix-sum problem). Although the depth can be poly-logarithmic, it is not work-efficient any more. If we set c to some given constant, the work and depth of this algorithm are $O(n\log n)$ and $O(n^{\epsilon})$ respectively.

9.3 2D Range Query

Given a set of n points in the 2D plane, a *range query* asks some information of points within the intersection of a horizontal range (x_L, x_R) and vertical range (y_L, y_R).

The 2D range query can be answered using a two-level map structure *RangeMap*, each level corresponding to one dimension of the coordinates. The structure can answer both counting queries and list-all queries. The definition (the outer map R_M with inner map R_I) and an illustration are shown in Table 9.2 and Figure 9.1 (a). In particular, the key of the outer map is the coordinate of each point and the value is the count. The augmented value of such an outer map, which is the inner map, contains the same set of points, but are sorted by y-coordinates. Therefore, the base function of the outer map is just a singleton on the point and the combine function is *Union*. The augmented value of the inner map counts the number of points in this (sub-)map. Then the construction of a sequence s of points can be done with the augmented map interface as: $r_M = R_M.\mathit{Build}(s)$.

To answer queries, we use two nested range searches (x_L, x_R) on the outer map and (y_L, y_R) on the corresponding inner map. The counting query can be represented using the augmented map interface as:

$$\begin{aligned}&\mathit{RangeQuery}(r_M, x_L, x_R, y_L, y_R) = \\ &\qquad R_I.\mathit{aug\_range}(R_M.\mathit{aug\_range}(r_M, x_L, x_R), y_L, y_R)\end{aligned} \tag{9.3}$$

The list-all query can be answered similarly using $R_I.\mathit{range}$ instead of $R_I.\mathit{aug\_range}$.

In this thesis, we use augmented trees for inner maps. We discuss two implementations of the outer map: the augmented tree, which yields a range-tree-like data structure, and the prefix structures, which yields a sweepline algorithm. We further discuss efficient updates on *RangeTree* using the augmented map interface.

9.3.1 2D Range Tree

If the outer map is supported using the augmented tree structure, the *RangeMap* becomes a range tree (*RangeTree*). In this case we do not explicitly build $R_M.\mathit{aug\_range}(r_M, x_L, x_R)$ in queries. Instead, as the standard range tree query algorithm, we search the x-range on the outer tree, and conduct the y-range queries on the related inner trees. This operation is supported by the function *aug_project* in the augmented map interface and the PAM library as follows.

$$\mathit{RangeQuery}(r_M, x_L, x_R, y_L, y_R) =$$

$$\mathit{let}\ g'(r_I) = \mathit{aug\_range}(r_I, y_L, y_R) \tag{9.4}$$

$$\mathit{in}\ \mathit{aug\_project}(g', +_W, r_M, x_L, x_R) \tag{9.5}$$

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' (i.e., $+_W$) 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 *aug_project* 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.

Such a tree structure can be constructed within work $O(n \log n)$ and depth $O(\log^3 n)$ (theoretically the depth can be easily reduced to $O(\log^2 n)$, but in the experiments we use the $O(\log^3 n)$ version to make fewer copies of data). It answers the counting query in $O(\log^2 n)$ time, and report all queried points in $O(k + \log^2 n)$ time for output size k.

Fast Updates on Range Trees Using Augmented Map Interface. The tree-based augmented map interface supports insertions and deletions (implementing the appropriate rotations). This can be used to insert and delete on the augmented tree interface. However, by default this requires updating the augmented values from the leaf to the root, for a total of $O(n)$ work. Generally, if augmented trees are used to support augmented maps, the insertion function will re-compute the augmented values of all the nodes on the insertion path, because inserting an entry in the middle of a map could completely change the augmented value. In the range tree, the cost is $O(n)$ per update because the combine function (*Union*) has about linear cost. To avoid this we implemented a version of "lazy" insertion/deletion that applies when the combine function is commutative. Instead of recomputing the augmented values it simply adds itself to (or removes itself from) the augmented values along the path using f and g. This is similar to the standard range tree update algorithm [195].

The amortized cost per update is $O(\log^2 n)$ if the tree is weight-balanced. Here we take the insertion as an example, but similar methodology can be applied to any mix of insertion and deletion sequences (to make deletions work, one may need to define the inverse function f^{-1} of the combine function). Intuitively, for any subtree of size m, imbalance occur at least every $\Theta(m)$ updates, each cost $O(m)$ to rebalance. Hence the amortized cost of rotations per level is $O(1)$, and thus the for a single update, it is $O(\log n)$ (sum across all levels). Directly inserting the entry into all inner trees on the insertion path causes $O(\log n)$ insertions to inner trees, each cost $O(\log n)$. In all the amortized cost is $O(\log^2 n)$ per update.

Similar idea of updating multi-level trees in (amortized) poly-logarithmic time can be applied to *SegTree**, *RecTree* and *RecTree**. For *SegTree*, the combine function is not communicative, and thus update may be more involved than simply using the interface of lazy-insert function.

9.3.2 The Sweepline Algorithm

We now present a parallel sweepline algorithm *RangeSwp* for 2D range queries using our parallel sweepline paradigm, which can answer counting queries efficiently. We use the prefix structures to represent the outer map. Then each prefix structure is an inner map tracking all points up to the current point sorted by y-coordinates. The combine function of the outer map is *Union*, so the update function h can be an insertion and the fold function ρ builds an inner map from a list of points. The definition of this sweepline paradigm R_S is shown in Table 9.2.

Since the inner maps are implemented by augmented trees, the theoretical bound of the functions (*insert*, *build*, and *union*) are consistent with the assumptions in Theorem 9.2.1. Thus the theoretical cost of *RangeSwp* follows from Theorem 9.2.1. This data structure takes $O(n \log n)$ space because of persistence. We note that previous work [127] showed a more space-efficient version (linear space), but the goal in this chapter is to show a generic paradigm that can easily implement many different problems without much extra effort. naively such a data structure would cost $O(n^2)$ space as the prefix structure sizes are 1 through n. However if persistence is supported by path copying, at most one new tree node is created in unit work. Thus the extra space it requires is asymptotically no more than its work, which is $O(n \log n)$. Note that in [127] a more space-efficient version (linear space) is shown, but our point here is to show that our paradigm is generic and simple for many different problems without much extra cost.

Answering Queries. Computing $ARange(r_M, x_L, x_R)$ explicitly in Equation 9.3 on *RangeSwp* can be costly. We note that it can be computed by taking a *Difference* on the prefix structure t_R at x_R and the prefix structure t_L at x_L (each can be found by a binary search). If only the count is required, a more efficient query can be applied. We can compute the number of points in the range (y_L, y_R) in t_L and t_R respectively, using *ARange*, and the difference of them is the answer. Two binary searches cost $O(\log n)$, and the range search on y-coordinates costs $O(\log n)$. Thus the total cost of a single query is $O(\log n)$.

Extension to Report All Points. This sweepline algorithm can be inefficient in list-all queries. Here we propose a variant for list-all queries. It is similar to *RangeSwp*, but instead of the count, the augmented value is the maximum x-coordinate among all points in the map. To answer queries we first narrow the range to the points in the inner map t_R by just searching x_R. In this case, t_R is an augmented tree structure. Then all queried points are those in t_R with x-coordinates larger than x_L and y-coordinate in $[y_L, y_R]$. We still conduct a standard range query in $[y_L, y_R]$ on t_R, but adapt an optimization that if the augmented value of a subtree node is less than x_L, the whole subtree is discarded. Otherwise, at least part of the points in the tree would be relevant and we recursively deal with its two subtrees.

The cost of this algorithm is similar to the *aug_filter* algorithm, which is $O(k \log(n/k + 1))$ on output k elements. In total, the cost of one query is $O(\log n + k \log(\frac{n}{k} + 1))$ for output size k. Comparing with *RangeTree*, which costs $O(k + \log^2 n)$ per query, this algorithm is asymptotically more efficient when $k < \log n$.

9.4 2D Segment Query

Given a set of non-intersecting 2D segments, and a vertical segment S_q, a segment query asks for some information about the segments that cross S_q. We define a segment as its two endpoints (l_i, r_i) where $l_i, r_i \in D, x(l_i) \leq x(r_i)$, and say it *starts from* $x(l_i)$ and *ends at* $x(r_i)$.

In this chapter, we introduce a two-level map structure *SegMap* addressing this problem (shown in Table 9.2 and Figure 9.1 (b)). The keys of the outer map are the x-coordinates of all endpoints of the input segments, and the values are the corresponding segments. Each (sub-)outer map corresponds to an interval on the x-axis (from the leftmost to the rightmost key in the sub-map), noted as the *x-range* of this map. The augmented value of an outer map is a pair of inner maps: $L(\cdot)$ (the *left open set*) which stores all input segments starting outside of its x-range and ending inside (i.e., only the right endpoint

Range Query:								
Inner Map:								
	R_I	=**AM**($\boldsymbol{K}$: D;	$\boldsymbol{\prec}$: $<_Y$; $\boldsymbol{V}$: $\mathbb{Z}$;	$\boldsymbol{A}$: $\mathbb{Z}$;	$\boldsymbol{g}$: $(k, v) \mapsto 1$;	$\boldsymbol{f}$: $+_{\mathbb{Z}}$;	$\boldsymbol{I}$: 0	)
Outer Maps:								
- RangeMap	R_M	=**AM**($\boldsymbol{K}$: D;	$\boldsymbol{\prec}$: $<_X$; $\boldsymbol{V}$: $\mathbb{Z}$;	$\boldsymbol{A}$: R_I;	$\boldsymbol{g}$: R_I.singleton;	$\boldsymbol{f}$: R_I.union;	$\boldsymbol{I}$: $\emptyset$	)
- RangeSwp	R_S	= **PS** ($\boldsymbol{P}$: D;	$\boldsymbol{\prec}$: $<_X$; $\boldsymbol{T}$: R_I;	$\boldsymbol{t_0}$: $\emptyset$	$\boldsymbol{h}$: R_I.insert	$\boldsymbol{\rho}$: R_I.build;	$\boldsymbol{f}$: R_I.union	)
Segment Query:								
Inner Map:								
	S_I	=**AM**($\boldsymbol{K}$: $D \times D$;	$\boldsymbol{\prec}$: $<_Y$; $\boldsymbol{V}$: $\emptyset$;	$\boldsymbol{A}$: $\mathbb{Z}$;	$\boldsymbol{g}$: $(k, v) \mapsto 1$;	$\boldsymbol{f}$: $+_{\mathbb{Z}}$;	$\boldsymbol{I}$: 0	)
Outer Maps:								
- SegMap	S_M	=**AM**($\boldsymbol{K}$: X;	$\boldsymbol{\prec}$: $<_X$; $\boldsymbol{V}$: $D \times D$;	$\boldsymbol{A}$: $S_I \times S_I$;	$\boldsymbol{g}$: g_{seg}	$\boldsymbol{f}$: f_{seg}	$\boldsymbol{I}$: $(\emptyset, \emptyset)$	)
	$\boldsymbol{g}_{\text{seg}}\boldsymbol{(k, (p_l, p_r))}$: $\begin{cases} (\emptyset, S_I.\text{singleton}(p_l, p_r), \text{when } k = x(p_l) \\ (S_I.\text{singleton}(p_l, p_r), \emptyset), \text{when } k = x(p_r) \end{cases}$, $\boldsymbol{f}_{\text{seg}}$: See Equation 9.6;							
- SegSwp	S_S	= **PS** ($\boldsymbol{P}$: $D \times D$;	$\boldsymbol{\prec}$: $<_X$; $\boldsymbol{T}$: S_I;	$\boldsymbol{t_0}$: $\emptyset$;	$\boldsymbol{h}$: h_{seg};	$\boldsymbol{\rho}$: ρ_{seg};	$\boldsymbol{f}$: f_{seg}	)
	$\boldsymbol{h}_{\text{seg}}\boldsymbol{(t, p)} = \begin{cases} S_I.\text{insert}(t, p), \text{when } p \text{ is a left endpoint} \\ S_I.\text{delete}(t, p), \text{when } p \text{ is a right endpoint} \end{cases}$,							
	$\boldsymbol{\rho}_{\text{seg}}(\langle p_i \rangle) = \langle L, R \rangle \begin{cases} L \in S_I\text{: segments with } \textit{right} \text{ endpoint in } \langle p_i \rangle \\ R \in S_I\text{: segments with } \textit{left} \text{ endpoint in } \langle p_i \rangle \end{cases}$							
Rectangle Query:								
Inner Map:								
	G_I	=**AM**($\boldsymbol{K}$: Y;	$\boldsymbol{\prec}$: $<_Y$; $\boldsymbol{V}$: $D \times D$;	$\boldsymbol{A}$: Y;	$\boldsymbol{g}$: g_i;	$\boldsymbol{f}$: $\max_Y$;	$\boldsymbol{I}$: $-\infty$	)
Outer Maps:								
- RecMap	G_M	=**AM**($\boldsymbol{K}$: X;	$\boldsymbol{\prec}$: $<_X$; $\boldsymbol{V}$: $D \times D$;	$\boldsymbol{A}$: $G_I \times G_I$;	$\boldsymbol{g}$: g_{rec}	$\boldsymbol{f}$: f_{rec}	$\boldsymbol{I}$: $(\emptyset, \emptyset)$	)
	g_{rec} and f_{rec} are defined similarly as g_{seg} and f_{seg}							
- RecSwp	G_S	= **PS** ($\boldsymbol{P}$: $D \times D$;	$\boldsymbol{\prec}$: $<_X$; $\boldsymbol{T}$: G_I;	$\boldsymbol{t_0}$: $\emptyset$;	$\boldsymbol{h}$: h_{rec};	$\boldsymbol{\rho}$: ρ_{rec};	$\boldsymbol{f}$: f_{seg}	)
	$\boldsymbol{g_i}(k, (p_l, p_r)) = y(p_r)$, g_{rec}, f_{rec}, h_{rec} and ρ_{rec} are defined similarly as g_{seg}, f_{seg}, h_{seg} and ρ_{seg}							

Table 9.2: Definitions of all structures in Chapter 9 for 2D geometric queries - Although this table seems complicated, we note that it *fully* defines all data structures in this chapter using augmented maps. X and Y are types of x- and y-coordinates. $D = X \times Y$ is the type of a point.

is in its x-range), and symmetrically $R(\cdot)$ (the *right open set*) with all segments starting inside but ending outside. We call them the *open sets* of the corresponding interval. The open sets of an interval u can be computed by combining the open sets of its sub-intervals. In particular, suppose u is composed of two contiguous intervals u_l and u_r, then u's open sets can be computed by a function f_{seg} as:

$$\begin{aligned} &f_{\text{seg}}(\langle L(u_l), R(u_l) \rangle, \langle L(u_r), R(u_r) \rangle) = \\ &\qquad \langle L(u_l) \cup (L(u_r) \backslash R(u_l)), R(u_r) \cup (R(u_l) \backslash L(u_r)) \rangle \end{aligned} \tag{9.6}$$

Intuitively, taking the right open set as an example, it stores all segments starting in u_r and going beyond, or those stretching out from u_l but *not* ending in u_r. This function is associative. We use f_{seg} as the combine function of the outer map.

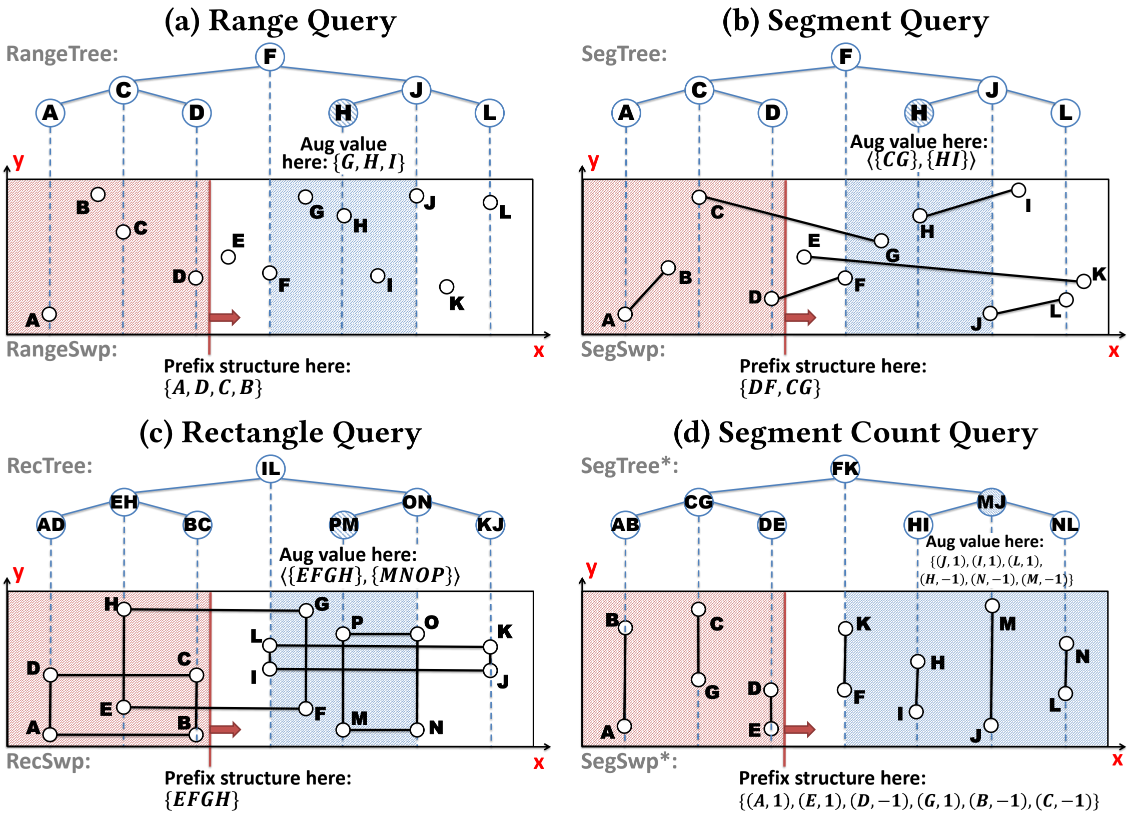

(a) Range Query
RangeTree:
F
C
J
A
D
H
L
Aug value here: $\{G, H, I\}$
y
B
C
D
A
E
G
H
J
L
F
I
K
x
RangeSwp:
Prefix structure here: $\{A, D, C, B\}$
(b) Segment Query
SegTree:
F
C
J
A
D
H
L
Aug value here: $\langle\{CG\}, \{HI\}\rangle$
y
C
B
A
D
E
F
G
H
I
K
J
L
x
SegSwp:
Prefix structure here: $\{DF, CG\}$
(c) Rectangle Query
RecTree:
IL
EH
ON
AD
BC
PM
KJ
Aug value here: $\langle\{EFGH\}, \{MNOP\}\rangle$
y
H
G
P
O
K
L
I
J
D
C
E
F
A
B
M
N
x
RecSwp:
Prefix structure here: $\{EFGH\}$
(d) Segment Count Query
SegTree*:
FK
CG
MJ
AB
DE
HI
NL
Aug value here: $\{(J, 1), (I, 1), (L, 1), (H, -1), (N, -1), (M, -1)\}$
y
B
C
G
D
E
A
K
F
H
I
M
J
N
L
x
SegSwp*:
Prefix structure here: $\{(A, 1), (E, 1), (D, -1), (G, 1), (B, -1), (C, -1)\}$

Figure 9.1: An illustration of all data structures based on augmented maps for geometric quereis - The input data are shown in the middle rectangle. We show the tree structures on the top, and the sweepline algorithm on the bottom. All the inner trees (the augmented values or the prefix structures) are shown as sets (or a pair of sets) with elements listed in sorted order.

The base function g_{seg} of the outer map (see Table 9.2) computes the augmented value of a single entry. For an entry $(x_k, (p_l, p_r))$, the interval it represents is the solid point at x_k. WLOG we assume $x_k = x(p_l)$ such that the key is the left endpoint. Then the only element in its open sets is the segment itself in its right open set. If $x_k > x_v$ it is symmetric.

We organize all segments in an inner map sorted by their y-coordinates and augmented by the count, such that in queries, the range search on y-coordinates can be done in the corresponding inner maps. We note that all segments in a certain inner tree must cross one common x-coordinate. For example, in the left open set of an interval i, all segments must cross the left border of i. Thus we can use the y-coordinates at this point to determine the ordering of all segments. Note that input segments are non-intersecting, so this ordering of two segments is consistent at any x-coordinate. The definition of such an inner map is in Table 9.2 (the inner map S_I). The construction of the two-level map *SegMap* (S_M) from a list of segments $B = \{(p_l, p_r)\}$ can be done as follows:

$$\begin{aligned} s_M = & S_M.build(B'), \text{ where} \\ & B' = \{(x(p_l), (p_l, p_r)), (x(p_r), (p_l, p_r)) : (p_l, p_r) \in B\} \end{aligned}$$

Assume the query segment is (p_s, p_e), where $x(p_s) = x(p_e) = x_q$ and $y(p_s) < y(p_e)$. The query will first find all segments that cross x_q, and then conduct a range query on $(y(p_s), y(p_e))$ on the y-coordinate among those segments. To find all segments that cross x_q, note that they are the segments starting before x_q but ending after x_q, which are exactly those in the right open set of the interval $(-\infty, x_q)$. This can be computed by the function *ALeft*. The counting query can be done using the augmented map interface as:

$$\begin{aligned} SegQuery(s_M, p_s, p_e) = & S_I.aug\_range \\ & (ROpen(S_M.aug\_left(s_M, x(p_t))), y(p_s), y(p_e)) \end{aligned}$$

where $ROpen(\cdot)$ extracts the right open set from an open set pair. The list-all query can be answered similarly using S_I*.Range* instead of S_I*.ARange*.

We use augmented trees for inner maps. We discuss two implementations of the outer map: the augmented tree (which yields a segment-tree-like structure *SegTree*) and the prefix structures (which yields a sweepline algorithm *SegSwp*). We also present another two-level augmented map (*Segment* Map*) structure that can answer counting queries on axis-parallel segments.

9.4.1 The Segment Tree

If the outer map is implemented by an augmented tree, the *SegMap* becomes very similar to a segment tree (noted as *SegTree*). Previous work has studied a similar data structure [17, 32, 101]. We note that their version can deal with more types of queries and problems, but we know of no implementation work of a similar data structure. Our goal is to show how to apply the simple augmented map framework to model the segment query problem, and show an efficient and concise parallel implementation of it.

In segment trees, each subtree represents an open interval, and the union of all intervals in the same level span the whole interval (see Figure 9.1 (b) as an example). The intervals are separated by the endpoints of the input segments, and the two children partition the interval of the parent. Our version is slightly different from the classic segment trees in that we also use internal nodes to represent a point on the axis. For example, a tree node u denoting an interval (l, r) have its left child representing $(l, k(u))$, right child for $(k(u), r)$, and the single node u itself, is the solid point at it key $k(u)$. For each tree node, the *SegTree* tracks the open sets of its subtree's interval, which is exactly the augmented value of the sub-map rooted at u. The augmented value (the open sets) of a node can be generated by combining the open sets of its two children (and the entry in itself) using Equation 9.6.

Answering Queries more efficiently. Calling the *ALeft* function on the outer tree of *SegTree* is costly, as it would require $O(\log n)$ function calls of *Union* and *Difference* on the way. Here we present a more efficient query algorithm making use of the tree structure, which is a variant of the algorithm in [32, 101]. Besides the open sets, in each node we store two helper sets (called the *difference sets*): the segments starting in its left half and going beyond the whole interval ($R(u_l)\backslash L(u_r)$ as in Equation 9.6), and vice versa ($L(u_r)\backslash R(u_l)$). We note that the calculation of the difference sets is not associative, but depends on the tree structure. These difference sets are the side-effect of computing the open sets. Hence in implementations we just keep them with no extra work. Suppose x_q is unique to all the input endpoints. The query algorithm first searches x_q outer tree. Let u be the current visited tree node. Then x_q falls in either the left or the right side of $k(u)$. WLOG, assume x_q goes right. Then all segments starting in the left half and going beyond the range of u should be reported because they must cover x_q. All such segments are in $R(lc(u))\backslash L(rc(u))$, which is in u's difference sets. The range search on y-coordinates will be done on this difference sets tree structure. After that, the algorithm goes down to u's right child to continue the search recursively. The cost of returning all related segments is $O(k + \log^2 n)$ for output size k, and the cost of returning the count is $O(\log^2 n)$.

9.4.2 The Sweepline Algorithm

If prefix structures are used to represent the outer map, the algorithm becomes a sweepline algorithm *SegSwp* (shown as S_S in Table 9.2). We store at each endpoint p the augmented value of the prefix of all points up to p. Because the corresponding interval is a prefix, the left open set is always empty. For simplicity we only keep the right open set as the prefix structure, which is all "alive" segments up to the current event point (a segment (p_l, p_r) is alive at some point $x \in X$ iff $x(p_l) \leq x \leq x(p_r)$).

Sequentially, this is a standard sweepline algorithm for segment queries—as the line sweeping through the plane, each left endpoint should trigger an insertion of its corresponding segment into the prefix structure while the right endpoints cause deletions. We note that this is also what happens when a single point is plugged in as u_r in Equation 9.6. We use our parallel sweepline paradigm to parallelize this process. In the batching step, we compute the augmented value of each block, which is the open sets of the corresponding interval. The left open set of an interval are segments with their right endpoints inside the interval, noted as L, and the right open set is those with left endpoints inside, noted as R. In the sweeping step, the prefix structure is updated by the combine function f_{seg}, but only on the right open set, which is equivalent to first taking a *Union* with R and then a *Difference* with L. Finally, in the refining step, each left endpoint triggers an insertion and each right endpoint causes a deletion. This algorithm fits the sweepline abstraction in Theorem 9.2.1, so the corresponding bound holds.

Answering Queries. The *ALeft* function on the prefix structure is just a binary search of x_q in the sorted list of x-coordinates. In that prefix structure all segments are sorted by y-coordinates, and we search the query range of y-coordinates on that. In all, a query for reporting all intersecting segments costs $O(\log n + k)$ (k is the output size), and a query on counting related segments costs $O(\log n)$.

9.4.3 Data Structures for Segment Counting Queries

In this section, we present a simple two-level augmented map *SegMap** structure to answer segment count queries (see the *Segment Count Query* in Table 9.2 and Figure 9.1 (d)). This map structure can only deal with axis-parallel input segments. For each input segment (p_l, p_r), we suppose $x(p_l) = x(p_r)$, and $y(p_l) < y(p_r)$. We organize the x-coordinates in the outer map, and deal with y-coordinates in the inner trees. We first look at the inner map. For a set of 1D segments, a standard solution to count the segments across some query point x_q is to organize all end points in sorted order, and assign signed flags to them as values: left endpoints with 1, and right endpoints with -1. Then the prefix sum of the values up to x_q is the number of alive segments. To efficiently query the prefix sum we can organize all endpoints as keys in an augmented map, with values being the signed flags, and augmented values adding values. We call this map the count map of the segments.

To extend it to 2D scenario, we use a similar outer map as range query problem. On this level, the x-coordinates are keys, the segments are values, and the augmented value is the count map on y-coordinates of all segments in the outer map. The combine function is *union* on the count maps. However, different from range maps, here each tree node represents *two* endpoints of that segment, with signed flags 1 (left) and -1 (right) respectively, leading to a different base function (g^*_{seg}).

We maintain the inner maps using augmented trees. By using augmented trees and prefix structures as outer maps, we can define a two-level tree structure and a sweepline algorithm for this problem respectively. Each counting query on the count map of size m can be done in time $O(\log m)$. In all, the rectangle counting query cost time $O(\log^2 n)$ using the two-level tree structure *SegTree**, and $O(\log n)$ time using the sweepline algorithm *SegSwp**.

We present corresponding definition and illustration on both the multi-level tree structure and the sweepline algorithm in Table 9.2 and Figure 9.1 (d).

9.5 2D Rectangle Query

Given a set of rectangles in two dimensions, a rectangle query asks for all rectangles containing a query point $p_q = (x_q, y_q)$. Each rectangle $C = (p_l, p_r)$, where $p_l, p_r \in D$, is represented as its left-top and right-bottom vertices. We say the interval $[x(p_l), x(p_r)]$ and $[y(p_l), y(p_r)]$ are the *x-interval* and *y-interval* of C, respectively.

The rectangle query can be answered by a two-level map structure *RecMap* (G_M in Table 9.2 and Figure 9.1 (c)), which is similar to the *SegMap* as introduced in Section

9.4. The outer map organizes all rectangles based on their x-intervals using a similar structure to the outer map of *SegMap*. The keys of the outer map are the x-coordinates of all endpoints of the input rectangles, and the values are the rectangles. The augmented value of a (sub-)outer map is also the *open sets* as defined in *SegMap*, which store the rectangles that partially overlap the x-range of this sub-map. The combine function is accordingly the same as the segment map.

Each inner map of the *RecMap* organizes the rectangles based on their y-intervals. All the y-intervals in an inner tree are organized in an *interval tree* (the term *interval tree* refers to different definitions in the literature. We use the definition in [118]). The interval tree is an augmented tree (map) structure storing a set of 1D intervals sorted by the left endpoints, and augmented by the maximum right endpoint in the map. Previous work [262] has studied implementing interval trees using the augmented map interface. It can report all intervals crossing a point in time $O(\log n + k\log(n/k + 1))$ for input size n and output size k.

RecMap answers the enclosure query of point (x_q, y_q) using a similar algorithm to *SegMap*. The query algorithm first finds all rectangles crossing x_q by computing the right open set R in the outer map up to x_q using *ALeft*, which is an interval tree. The algorithm then selects all rectangles in R crossing y_q by applying a list-all query on the interval tree.

Using interval trees as inner maps does not provide an efficient interface for counting queries. We use the same inner map as in *SegMap** for counting queries.

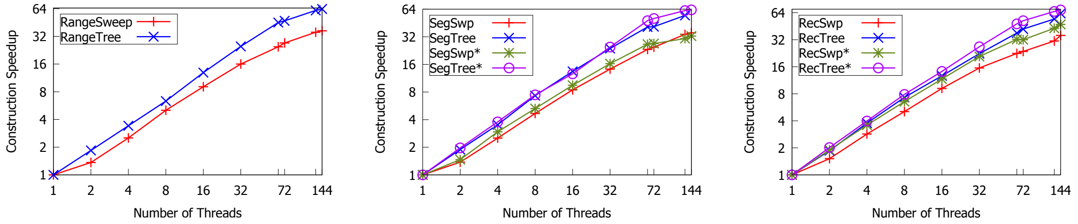
We use augmented trees for inner maps (the interval trees). We discuss two implementations of the outer map: the augmented tree (which yields a multi-level tree structure) and the prefix structures (which yields a sweepline algorithm).

9.5.1 The Multi-level Tree Structure.

RecMap becomes a two-level tree structure if the outer map is supported by an augmented tree, which is similar to the segment tree, and we use the same trick of storing the difference sets in the tree nodes to accelerate queries. The cost of a list-all query is $O(k\log(n/k + 1) + \log^2 n)$ for output size k.

9.5.2 The Sweepline Algorithm

If we use prefix structures to represent the outer map, the algorithm becomes a sweepline algorithm (G_S in Table 9.2). The skeleton of the sweepline algorithm is the same as *SegSwp*—the prefix structure at event point x stores all “live” rectangle at x. The combine function, fold function and update function are defined similar as in *SegSwp*, but onto inner maps as interval trees. This algorithm also fits the sweepline abstraction in Theorem 9.2.1, so the corresponding bound holds.



(a). Range Query **(b). Segment Query** **(c). Rectangle Query**

Figure 9.2: The speedup of building various data structures for range, segment and rectangle queries– $n = 10^8$.

To answer the list-all query of point (x_q, y_q), the algorithm first finds the prefix structure t_q at x_q, and applies a list-all query on the interval tree t_q at point y_q. The cost is $O(\log n + k\log(n/k+1))$ per query for output size k.

9.5.3 Data Structures for Rectangle Counting Queries

In this section, we extend the *RecMap* structure to *RecMap** for supporting fast counting queries. We use the exactly outer map as *RecMap*, but use base and combine functions on the corresponding inner maps. The inner map, however, is the same map as the *count map* in *SegMap** (S_I^* in Table 9.2). Then in queries, the algorithm will find all related inner maps, which are count maps storing all y-intervals of related rectangles. To compute the count of all the y-intervals crossing the query point y_q, the query algorithm simply apply an *ALeft* on the count maps.

We maintain the inner maps using augmented trees. Using augmented trees and prefix structures as outer maps, we can define a two-level tree structure and a sweepline algorithm for this problem respectively. The rectangle counting query cost time $O(\log^2 n)$ using the two-level tree structure *RecTree**, and $O(\log n)$ time using the sweepline algorithm *RecSwp**.

We present corresponding definition and illustration on both the multi-level tree structure and the sweepline algorithm in Table 9.2. The outer representation of *RecMap** is of the same format as *RecMap* as shown in Figure 9.1 (c).

9.6 Experiments

We implement all data structures in this chapter using PAM. We run our experiments on a 72-core Dell R930 with 4 x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache) with 1TB memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. Our code was compiled using the g++ 5.4.1 compiler which supports the Cilk Plus extensions. We compile with `-O2`.

All implementations of augmented trees used in the our algorithms are supported by PAM [262]. We implement the abstract parallel sweepline paradigm as described in

Section 9.2.2. On top of them, each of our data structures only need about 100 lines of code. Our code is available online at `https://github.com/cmuparlay/PAM`. More details about PAM can be found online [261] and in our previous work [262].

For range queries, we test *RangeTree* and *RangeSwp* in Section 9.3. The tested *RangeTree* implementation is based on the range tree code in the previous paper [262]. For segment queries, we test *SegTree* and *SegSwp* in Section 9.4, as well as the counting versions *SegTree** and *SegSwp** in Appendix 9.4.3. For rectangle queries, we test *RecTree* and *RecSwp* in Section 9.5, as well as the counting versions *RecTree** and *RecSwp** in Appendix 9.5.3. We use integer coordinates. We test both construction time and query time, and both counting queries and list-all queries. In some of the problems, the construction of the data structure for counting and list-all queries can be slightly different, and we always report the version for counting queries. On list-all queries, since the cost is largely affected by the output size, we test a small and a large query window with average output size of less than 10 and about 10^6 respectively. We accumulate query time over 10^3 large-window queries and over 10^6 small window queries. For counting queries we accumulate 10^6 queries. For parallel queries we process all queries in parallel using a parallel for-loop. The sequential algorithms tested in this chapter are directly running the parallel algorithms on one core. We use n for the input size, k the output size, p the number of threads. For the sweepline algorithms we set $b = p$, and do not apply the sweepline paradigm recursively to blocks.

We compare our sequential versions with two C++ libraries CGAL [98] and Boost [5]. CGAL provides a range tree [215] structure similar to ours, and the segment tree [215] in CGAL implements the 2D rectangle query. Boost provides an implementation of R-trees, which can be used to answer range, segment and rectangle queries. CGAL and Boost only support list-all queries. We parallelize the queries in Boost using OpenMP. CGAL uses some shared state in queries so the queries cannot be parallelized trivially. We did not find comparable parallel implementations in C++, so we compare our parallel query performance with Boost. We also compare the query and construction performance of our multi-level tree structures and sweepline algorithms with each other, both sequentially and in parallel.

In the rest of this section we show results for range, segment and rectangle queries and comparisons across all tested structures. We show that our implementations achieve good speedup (32-126x on 72 cores with 144 hyperthreads). The overall sequential performance (construction and query) of our implementations is comparable or outperforms existing implementations.

9.6.1 2D Range Queries

We test *RangeTree* and *RangeSwp* for both counting and list-all queries, sequentially and in parallel. We test 10^8 input points generated uniformly randomly. For counting queries, we generate endpoints of the query rectangle uniformly randomly. For list-all queries with large and small windows, we control the output size by adjusting the average

Algorithm	Build, s			Counting, μs			List-all (small), μs			List-all (large), ms		
	Seq.	**Par.**	**Spd.**	**Seq.**	**Par.**	**Spd.**	**Seq.**	**Par.**	**Spd.**	**Seq.**	**Par.**	**Spd.**
RangeSwp	243.89	7.30	33.4	12.74	0.15	86.7	11.44	0.13	85.4	213.27	1.97	108.4
RangeTree	200.59	3.16	63.5	61.01	0.75	81.1	17.07	0.21	80.5	44.72	0.69	65.2
Boost	315.34	-	-	-	-	-	25.41	0.51	49.8	1174.40	22.42	52.4
CGAL	525.94	-	-	-	-	-	153.54	-	-	110.94	-	-
SegSwp	254.49	7.20	35.3	6.78	0.09	75.3	6.18	0.08	77.2	255.72	2.65	96.5
SegTree	440.33	6.79	64.8	50.31	0.70	71.9	39.02	0.48	81.7	123.26	1.99	61.9
Boost	179.44	-	-	-	-	-	7421.30	113.09	65.6	998.20	23.21	43.0
SegSwp*	233.19	7.16	32.6	7.44	0.11	67.6	-	-	-	-	-	-
SegTree*	202.01	3.21	63.0	33.58	0.40	83.8	-	-	-	-	-	-
RecSwp	241.51	6.76	35.7	-	-	-	8.34	0.10	83.4	575.46	5.91	97.4
RecTree	390.98	6.23	62.8	-	-	-	43.57	0.58	75.1	382.26	5.35	71.4
Boost	183.65	-	-	-	-	-	52.22	0.94	55.6	706.50	11.10	63.6
CGAL[1]	398.44	-	-	-	-	-	90.02	-	-	4412.67	-	-
RecSwp*	585.18	12.37	47.32	6.56	0.05	126.1	-	-	-	-	-	-
RecTree*	778.28	11.34	68.63	39.75	0.35	113.6	-	-	-	-	-	-

Table 9.3: The running time of all data structures on geometric applications - "Seq.", "Par." and "Spd." refer to the sequential, parallel running time and the speedup. [1]: Result of CGAL is shown as on input size 5×10^6. On 5×10^7, CGAL did not finish in one hour.

length of the edge length of the query rectangle. We show the running time in Table 9.3. We show the scalability curve for construction in Figure 9.2 (a).

Sequential Construction. *RangeTree* and *RangeSwp* have similar performance and outperform CGAL (2x faster) and Boost (1.3-1.5x). Among all, *RangeTree* is the fastest in construction. We guess the reason of the faster construction of our *RangeTree* than CGAL is that their implementation makes copies the data twice (once in merging and once to create tree nodes) while ours only copies the data once.

Parallel Construction. *RangeTree* achieves a 63-fold speedup on $n = 10^8$ and $p = 144$. *RangeSwp* has relatively worse parallel performance, which is a 33-fold speedup, and 2.3x slower than *RangeTree*. This is likely because of its worse theoretical depth ($\tilde{O}(\sqrt{n})$ vs. $O(\log^2 n)$). As for *RangeTree*, not only the construction is highly-parallelized, but the combine function (*Union*) is also parallel. Figure 9.2(a) shows that both *RangeTree* and *RangeSwp* scale up to 144 threads.

Query Performance. In counting queries, *RangeSwp* shows the best performance in both theory and practice. On list-all queries, *RangeSwp* is much slower than the other two range trees when the query window is large, but shows better performance for small windows. These results are consistent with their theoretical bounds. Boost's R-tree is 1.5-26x slower than our implementations, likely because of lack of worst-case theoretical guarantee in queries. Our speedup numbers for queries are above 65 because they are embarrassingly parallel, and speedup numbers of our implementations are slightly higher than Boost.

9.6.2 2D Segment Query

We test 5×10^7 input segment, using *SegSwp*, *SegTree*, *SegSwp** and *SegTree**. Note that for these structures on input size n (number of segments), $2n$ points are needed in the map. Thus we use input size of $n = 5 \times 10^7$ for comparison with the maps for range queries. The x-coordinate of each endpoint is generated uniformly randomly. To guarantee that the input segments do not intersect, we generate n non-overlapping intervals as the y-coordinates and assign each of them to a segment uniformly randomly. For *SegSwp** and *SegTree**, each segment is directly assigned a y-coordinate uniformly randomly. For counting queries, we generate endpoints of the query segment uniformly randomly. For list-all queries with large and small windows, we control the output size by adjusting the average length of the query segment. We show the running times in Table 9.3. We also show the parallel speedup for construction in Figure 9.2(b).

Sequential Construction. Boost is 1.4x faster than *SegSwp* and 2.4x than *SegTree*. This is likely due to R-tree's simpler structure. However, Boost is 4-1200x slower in queries than our implementations. *SegTree* is the slowest in construction because it stores four sets (the open sets and the difference sets) in each node, and calls two *Difference* and two *Union* functions in each combine function.

Parallel Construction. In parallel construction, *SegTree* is slightly faster than *SegSwp*. Considering that *SegTree* is 1.7x slower than *SegSwp* sequentially, the good parallelism comes from its good scalability (64x speedup). The lack of parallelism of *SegSwp* is for the same reason as *RangeSwp*.

Query Performance. In the counting query and list-all query on small window size, *SegSweep* is significantly faster than *SegTree* as would be expected from its better theoretical bound. As for list-all on large window size, although *SegTree* and *SegSwp* have similar theoretical cost (output size k dominates the cost), *SegTree* is faster than *SegSwp* both sequentially and in parallel. This might have to do with locality. In the sweepline algorithms, the tree nodes even in one prefix structure were created at different times because of path-copying, and thus are not contiguous in memory, leading to bad cache performance. Both *SegSwp* and *SegTree* have better query performance than Boost's R-tree (8.7-1400x faster in parallel). Also, the Boost R-tree does not take advantage of smaller query windows. Comparing the sequential query performance on large windows with small windows, on outputting about 10^6x less points, *SegTree* and *SegSwp* are 3000x and 40000x faster respectively, while Boost's R-tree is only 130x faster. Our implementations on small windows is not 10^6x as fast as on large windows because on small windows the $\log n$ or $\log^2 n$ term dominates the cost. This illustrates that the bad query performance of R-trees comes from lack of worst-case theoretical guarantee. The query speedup of our implementations is over 61.

9.6.3 2D Rectangle Query

We test rectangle queries using *RecSwp*, *RecTree*, *RecSwp** and *RecTree**, on $n = 5 \times 10^7$. The the query points are generated uniformly randomly. For counting queries, the endpoints of the input rectangles are generated uniformly randomly. For list-all queries with large and small windows, we control the output size by adjusting the average length of the edge length of the input rectangle. The running times are reported in Table 9.3, and the parallel speedup for construction are in Figure 9.2(c).

Sequential Construction. Sequentially, *RecSwp*, *RecTree* and Boost R-tree have performance close to segment queries – Boost is faster in construction than the other two (1.6-2.1x), but is much slower in queries, and *RecTree* is slow in construction because of its more complicated structure. CGAL did not finish construction on $n = 5 \times 10^7$ in one hour, and thus we present the results on $n = 5 \times 10^6$. In this case, CGAL has a performance slightly worse than our implementations even though our input size is 10x larger.

Parallel Construction. The parallel performance is similar to the segment queries, in which *RecTree* is slightly faster than *RecSwp* because of good scalability (62x speedup).

Query Performance. In list-all queries on a small window size, *RecSwp* is significantly faster than other implementations due to its better theoretical bound. Boost is 1.2-9x slower than our implementations when query windows are small, and is 1.2-2x slower when query windows are large, both sequentially and in parallel. The query speedup of our implementations is over 71.

9.6.4 Discussion on the Data Structures for Counting Queries

We list the performance of the four data structures for fast answering counting queries in Table 1.3. *SegTree** and *SegSwp** both have faster construction time than *SegTree* and *SegSwp* probably because they have simpler structure and less functionality (cannot answer list-all queries). Another reason is that *SegTree** and *SegSwp** both have smaller outer map sizes (5×10^7 vs. 10^8), thus requiring fewer invocations of combine functions on the top level. *RecSwp** and *RecTree**, however, is about 2x slower than *RecSwp* and *RecTree*. This is because they have twice as large the inner tree sizes—an inner tree of a *RecMap* is an interval tree storing each rectangle once as its y-interval, while an inner tree of a *RecMap** is a count map storing each rectangle twice as the two endpoints of its y-interval.

Overall, the results of these four data structures consists with the other data structures. In constructions, the sweepline algorithms have better sequential performance, but the two-level tree structures have better speedup and parallel performance. In counting queries, the sweepline algorithms are always much faster than the two-level tree structures because of their better theoretical bound.

Algorithm	***m***	**Seq.** (s)	**Par.** (s)	**Spd.**
AugInsert	10	406.079	5.366	75.670
LazyInsert	10^5	12.506	-	-

Table 9.4: The performance of insertions on range trees using the lazy-insert function in PAM - "Seq.", "Par." and "Spd." mean the sequential running time, parallel running time and the speedup number respectively.

9.6.5 Experimental Results on Dynamic Range Trees

We use the lazy-insert function (assuming a commutativity combine function) in PAM to support the insertion on range trees and test the performance. We first build an initial tree of size using our construction algorithm, and then conduct a sequence of insertions on this tree. We compare it with the plain insertion function (denoted as *AugInsert*) in PAM which is general for the augmented tree (re-call the combine function on every node on the insertion path). We show the results in Table 9.4. Because the combine function takes linear time, each *AugInsert* function costs about 40s, meaning that even 5 insertions may cost as expensive as re-build the whole tree. This function can be parallelized, with speedup at about 75x. The parallelism comes from the parallel combine function *Union*. The *LazyInsert* function is much more efficient and can finish 10^5 insertions in 12s sequentially. Running in parallel does not really help in this case because the the combine function (*Union*) is very rarely called, and even when it is called, it would be on very small tree size. When the size increases to 10^6, the cost is also greater than rebuilding the whole tree. This means that in practice, if the insertions come in large bulks, rebuilding the tree (even sequentially) is often more efficient than inserting each point one by one into the tree. When there are only a small number of insertions coming in streams, *LazyInsert* is reasonable efficient.

9.6.6 Experimental Results on Space Consumption

In Table 9.5, we report the space consumption using our data structures of range, segment and rectangle queries as examples to show the space-efficiency of our implementations. We note that for rectangle queries, the outer map structure is similar to segment queries, and thus can be estimated roughly using the results of segment queries. We store in each tree node a key, a value, an augmented value, the subtree size, two pointers to its left and right children and a reference counter for efficient garbage collection. Each inner tree is represented using a root pointer.

For all of them we estimate the theoretical number of nodes needed and show them in the table. The theoretical space cost is $O(n \log n)$ for all of them. For *SegTree* we use $2n \log n$ to estimate the number of inner tree nodes, and for the rest of them we simply use $n \log n$. This is because in a *SegTree*, there are 4 inner trees stored in each of the outer tree node, and in the worst case, a segment can appear in at most two of them (one in the open sets and the other in difference sets). We compute the ratio as the actual used inner tree nodes divided by the theoretical number of inner nodes. All results in Table 9.5 are from experiments on all 144 threads.

As shown in the table, the two multi-level trees have ratio less than 100%. This saving is mostly from the path-copying for supporting the persistence. In other words, in the process of combining the inner trees, some small pieces are preserved, and are shared among multiple inner trees. This phenomenon should be more significant when the input distribution is more skewed. In our case, because of our input is selected uniformly at random, the saving ratio is about 10%-15%. One special case is the *SegTree*, where the ratio is only about 50%. This is because even though theoretically in the worst case, each segment can appear in the augmented values of $O(\log n)$ outer tree nodes (one per level), in most of the cases they cancel out in the combine function. As a result, the inner tree sizes can be very small especially when the segments are short. As shown in the table, the actual number of required inner tree nodes is only about a half of the worst case, when the input endpoints are uniformly distributed.

For the sweepline algorithms, the actual used nodes are often slightly more than estimated. This is because in the parallel version of our sweepline paradigm, the trees at the beginning of each block are built separately. In the batching step, $O(n \log b)$ new tree nodes are created because of the b *Union* functions. In the sweeping step, $n \log n + O(n)$ new nodes are created. Because of the fold-and-sweep parallel sweepline paradigm we are using, we waste some space in the second step, when constructing the prefix structures at the beginning of each block. As a very rough estimation, we waste about $n \log b$ (off by a small constant) nodes. In our experiment, $b = 144$, $\log b \approx 7.2$, $\log n \approx 26$, which means that we may have a factor of $\log n / \log b \approx 27\%$ waste of tree nodes. This roughly matches our result of *RangeSwp*. For *SegSwp*, the nodes are inserted and then deleted at some point, and thus the size of the prefix structures can be small for most of the time. In this case the wasted number of inner tree nodes is much fewer, which is only about 17%.

In all, all of the tested data structures on range and segment tests use less than than $1.5n \log n$ tree nodes. Even the largest of them only cost 130G memory for input size 10^8, which includes all costs of storing keys and values, as well as pointers and other information in tree nodes.

9.6.7 Summary

The sweepline algorithms usually perform better in sequential construction, but in parallel are slower than two-level trees. This has to do with the better scalability of the two-level trees. With properly defined augmentation, the construction of the two-level trees is automatically done by the augmented map constructor in PAM, which is highly-parallelized (polylog depth). For the sweepline algorithms, the parallelism comes from the blocking-and-batching process, with a $\tilde{O}(\sqrt{n})$ depth. Another reason is that more threads means more blocks for sweepline algorithms, introducing more overhead in batching and folding.

	# of Outer Nodes ($\times 10^6$)	Size of an Outer Node (Bytes)	# of Inner Nodes ($\times 10^9$)	Size of an Inner Node (Bytes)	Theoretical # of Inner Nodes ($\times 10^9$)	Ratio (%)	Used Space (G bytes)
RangeTree	99.97	48	2.29	40	2.56	89.5	89.75
RangeSwp	-	-	3.52	40	2.56	137.5	130.99
SegTree	100.00	80	2.84	40	5.32	53.5	113.56
SegSwp	-	-	3.01	40	2.56	117.7	112.13

Table 9.5: The space consumption of our data structures on range and segment queries - The theoretical number of inner nodes are estimates as $2n \log n$ for *SegTree*, and $n \log n$ for the rest of them. The ratio is computed as the actual used inner nodes / the theoretical number of inner nodes.

Most of the implementations have close construction time. Sequentially *SegTree* and *RecTree* are much slower than the others, because they store more information in each node and have more complicated combine functions. The speedup of all sweepline algorithms are close at about 30-35x, and all two-level trees at about 62-68x.

In general, the sweepline algorithms are better in counting queries and small window queries but are slower in large window queries. In counting queries and small window queries (when the output size does not dominate the cost) the sweepline algorithms perform better because of the better theoretical bound. On large window queries (when the output size dominates the cost), the two-level tree structures performs better because of better locality.

Our implementations scale to 144 threads, achieve good speedup and show efficiency on large input size. Our implementation show good performance both sequentially and in parallel on input size as large as 10^8. On 72 cores with 144 hyperthreads, the construction speedup is 32-69x, and the query speedup is 65-126x.

Our implementations are always faster on queries than existing implementations with comparable or faster construction time, even sequentially. Our implementations outperform CGAL in both construction and queries by at least 2x. Overall, the Boost R-tree is about 1.5x slower to 2.5x faster than our algorithms in construction, but is always slower (1.6-1400x) in queries both sequentially and in parallel, likely because of lack of worst-case theoretical guarantee of R-trees.

Chapter 10

A Tree for Concurrent Reads and Writes

10.1 Introduction

One potential issue of path-copying-based data structures is that concurrent operations on the tree does not come into effect on the same version. In particular, any concurrent thread only update its local snapshot. For a DBMS although SI is supported, there is not guarantee for supporting serializability for concurrent update transactions [97]. A simple solution is to only allow a single writer to sequentialize all *update*s (e.g., flat-combining [155]). This is likely to be adequate if updates are large with significant internal parallelism, or if the rate of smaller updates is light (in the order of tens of thousands per second), as might be the case for a data structure or a system dominated by analytical queries. It is unlikely to be adequate for a data structure or a system with high update rates of small transactions. The DBMS Hyder [238], which uses multiversioning with path copying, addresses this by allowing transactions to proceed concurrently and then merging the copied trees using a "meld" operation. The DBMS, however, must sequentialize these melding steps so that it creates new versions one at a time. The melding process can also cause an abort even when the updates are logically independent—for example when the first update does a rotation on an internal tree node that the second visits.

Another approach is to batch updates as part of a group commit operation [121]. The basic approach is for the data structure or the system to process a set of updates obeying a linear order. It then detects any logical conflicts based on this order and the operations they performed (e.g., write-read conflicts). The system next removes any conflicted updates and then commits the remaining conflict-free updates as a batch. This approach is taken by a variety of systems, including Calvin [266], FaunaDB [1], PALM [247], and BATCHER [18]. The advantage of this approach is that the batch update can make use of parallelism [1, 18], consensus in only needed at the granularity of batches in distributed systems [1, 266], and the batched updates can make more efficient use of the disk in disk-based systems [30]. The challenge of the approach, however, is in detecting conflicts; all of the above systems perform this step differently.

For simple point updates, this thesis proposes to use *batching*. This means to only use one global writer to collect all the concurrent updates, coordinate internally, and commit altogether in parallel. Because of the parallel bulk operations supported by P-Trees using divide-and-conquer, such bulk operations (e.g., *multi_insert*) avoids conflict and contention, which is likely to be more efficient than invoking all operations concurrently. Similar approach has also been shown to be efficient in practice in previous work [1, 18, 155, 247, 266].

Theoretically, P-Trees with batching yields a concurrent data structure allowing for lock-free writes, such as insertion, deletion and updates, as well as wait-free read-only queries, such as searching and range queries.

Experiments show that using the P-Tree with batching is efficient on workloads of concurrent updates and searching with different read-write ratios. We use four workloads in YCSB (Yahoo! Cloud Serving Benchmark). On all tested workloads, with reasonable latency, P-Trees outperforms state-of-the-art concurrent data structures which even do not support snapshotting.

10.2 Approach

In our system we use batching and use multi-insert and multi-delete to apply batches of point updates. As discussed, these primitives have significant internal parallelism. In this chapter we do not consider how to detect conflicts for arbitrary transactions (previous work could help here), but study the approach for simple point transactions such as in the YCSB benchmark (insertions and deletions). In this case, the only conflicts are between updates on the same key, and keeping the last update on the key is sufficient.

To batch a set of updates, we wait for some amount of time, allowing any updates to accumulate in a buffer, and then apply all the updates in the buffer together as a batch in parallel. While processing the batch new updates can accumulate in another buffer.

10.2.1 The Batching Algorithm

For the batching algorithm, we use a simple strategy, where each process is allocated a buffer array with a head and a tail index. Each process submits all its updates to the buffer by adding them to the tail. Periodically, the writer goes over each array, assembles all operations between the current head and tail into the batch, and then moves the head index to the current tail index (plus one). There is no contention between processes because each reader only operates its own buffer at the tail, and the single writer only operate on the head index of all buffers. The batching process also works in parallel using a parallel packing.

```
int head[P], tail[P], offset[P];
void update(operation k, size_t id) {
  buffer[id].push_back(k); }

void single_writer() {
  while (true) {
    while (not enough operations && not a long time) {}
    int start[P], end[P], offset[P];
    for (int i = 0; i < P; i++) {
      start[i] = head[i]; end[i] = tail[i];
      int block = end[i]-start[i];
      if (block > max_op_each) {
        end[i] = start[i]+max_op_each;
        block = max_op_each;
      }
      offset[i+1] = offset[i]+block;
      head[i] = end[i]; }
    int m = offset[P];
    entry_type* a = new entry_type[m];

    parallel_for (int i = 0; i < P; i++) {
      par_for(int j=start[i]; j<end[i]; j++) {
        int ind = offset[i]+j-start[i];
        a[ind] = buffer[i][k]; } }
    d = commit_batch(d, a, m); }
}
```

Figure 10.1: The batching algorithm used in our system.

The updates are then committed to the data structure in a batch in parallel, using *multi_insert*. Our approach also allows each batch to be committed *atomically*, since committing the new version root makes all new tuples visible atomically. For more complicated transactions, we need to first build the dependency between transactions.

10.2.2 Latency-throughput Tradeoff

In batching there is a tradeoff between throughput and latency, which is controlled by the batching size. The batching size reflects the "granularity" of batching. For improving throughput, larger batching size is more favorable because of two reasons. First, larger batching sizes hide the amortized overhead of dealing with each operation, including sorting, combining operations to the same key, etc. Secondly, larger batching sizes allow us to exploit more parallelism. On the other hand, larger batch sizes cause each operation to wait longer. There is a long *latency* for the first operation coming to the batch between when it is invoked and when it comes into effect. This is because (1) it has to wait for more operations until a batch is large enough to be committed, and (2) a larger batch requires a longer time to execute. This is harmful to settings that do not tolerate long latency. Therefore, there is a tradeoff between throughput and latency—to get a certain throughput, how much latency the system has to tolerate.

In our implementation, we control the latency to be at the same magnitude of network latency, such that the latency waiting for a batch to finish does not dominate the cost. This tradeoff is analyzed in Section 10.3.2.

10.3 Experiments

10.3.1 Workloads

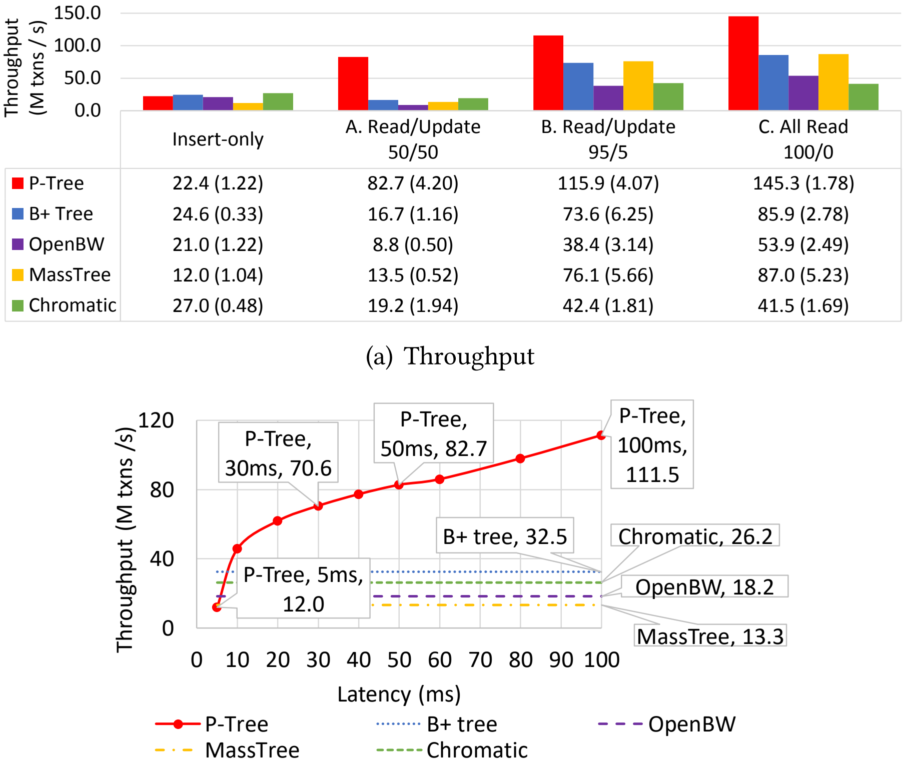
YCSB. We test YCSB workloads **Insert-only** (constructing the tree with parallel insertions), **A** (read/update, 50/50), **B** (read/update, 95/5), and **C** (read-only) with Zipfian distributions. The database contains a single table with 50m entries using 64-bit integers as keys and values. Each workload contains 10^7 transactions. Our DBMS executes read transactions concurrently on the last committed snapshot in the database. We buffer the write transactions using the batching algorithm in [48, 50] and then commit them to the last snapshot of the database using *multi_insert* without blocking readers. We execute batched updates with the `replace` combine function (see Section 11.2). If there are multiple updates on the same key in a batch, then the last write persists. We report the mean of throughput across five trials and the standard deviation.

As described in Section 11.2.1, batching performance is highly affected by the batch granularity. On the one hand, we need to make batches reasonably small and frequent such that each query get a timely response. On the other hand, larger and less frequent batches, although causes long latency, usually lead to less overhead and better throughput. As a result, the appropriate batching size achieves a tradeoff between latency and throughput, which depends on the platform and the machine. We will show some in-depth study of the correlation between latency and throughput. Except for the experiments on latency-throughput tradeoff, which varies the latency, we control the latency to be 50 ms, which is the same magnitude of network latency, and thus is unlikely to dominate the cost. As a result, 50 ms is the acceptable limit in many application domains (e.g., internet advertising, on-line gaming).

10.3.2 Experiments

In this first experiment, we compare P-Trees with four other state-of-the-art concurrent indexes for in-memory DBMSs:

- **B+tree**. A memory-optimized B+tree using optimistic locking coupling [189].
- **Bw-Tree**. Microsoft's latch-free B+tree index from Hekaton [190]. We use the OpenBw-Tree implementation [268].
- **MassTree**. A hybrid B+tree that uses tries for nodes from the Silo DBMS [198].



	Insert-only	A. Read/Update 50/50	B. Read/Update 95/5	C. All Read 100/0
🟥 P-Tree	22.4 (1.22)	82.7 (4.20)	115.9 (4.07)	145.3 (1.78)
🟦 B+ Tree	24.6 (0.33)	16.7 (1.16)	73.6 (6.25)	85.9 (2.78)
🟪 OpenBW	21.0 (1.22)	8.8 (0.50)	38.4 (3.14)	53.9 (2.49)
🟨 MassTree	12.0 (1.04)	13.5 (0.52)	76.1 (5.66)	87.0 (5.23)
🟩 Chromatic	27.0 (0.48)	19.2 (1.94)	42.4 (1.81)	41.5 (1.69)

(a) Throughput

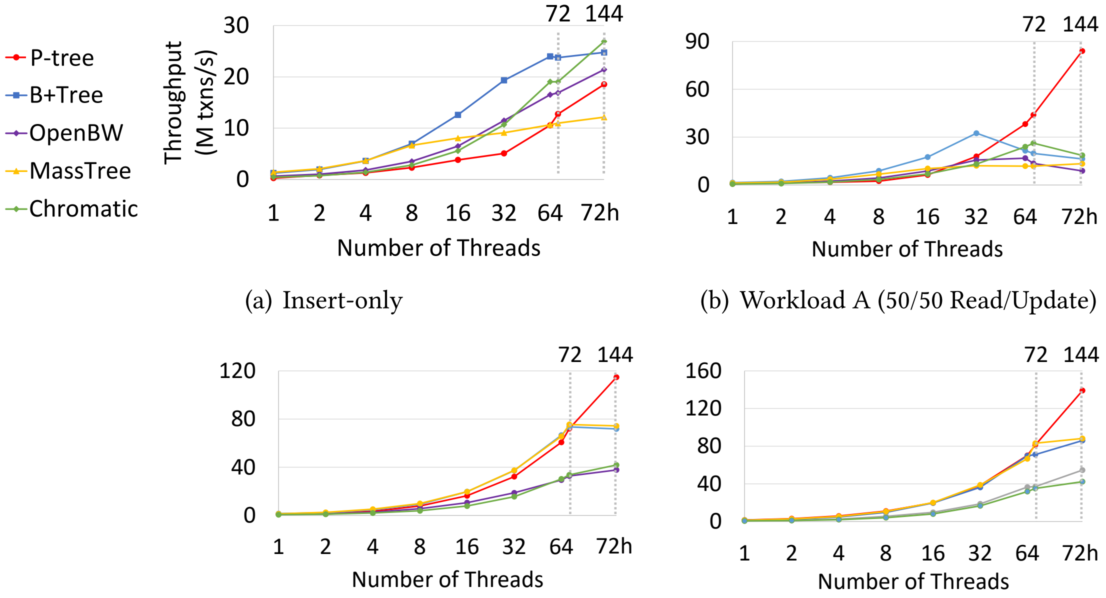
(b) Throughput-Latency Correlation (Workload A)

Figure 10.2: YCSB Workload Performance – Comparison of concurrent data structures for the YCSB workloads using 144 threads. (10.2(a)) Throughput measurements for each data structure. Numbers in the parentheses are the standard deviations. P-Trees execute concurrent updates in batches within 50 ms latency. (10.2(b)) The correlation between latency and throughput on YCSB workload A. The horizontal lines show the maximum throughput of the other data structures across different numbers of threads.

- **Chromatic Tree**. A lock-free Chromatic tree implementation in C++ [92, 96, 218].

The implementation of B+tree, OpenBw-Tree and MassTree are from Wang et al. [267, 268]. The Chromatic Tree implementation is from Brown et al. [93, 96]. We note that none of them support SI.

We use three experiments to evaluate P-Tree's performance on the OLTP benchmark YCSB. We first compare the throughput of all tested data structures on four workloads to understand the parallel performance of these data structures under different read/write ratios. We then experiment on the tradeoff between latency and throughput for P-Trees, using workload A as an example. Finally, we discuss the scalability curve for all tested data structures on all four workloads.



(a) Insert-only

(b) Workload A (50/50 Read/Update)

(c) Workload B (95/5 Read/Update)

(d) Workload C (Read-only)

Figure 10.3: YCSB Workload Scalability – We always control the latency to be ~50 ms for P-Trees. "72h" means 144 threads (with hyperthreading).

Performance. We first measure the throughput of all data structures. The numbers reported on P-Trees are within 50 ms latency. Figure 10.2(a) shows that P-Trees outperform or is competitive to the other implementations. P-Trees' standard deviation is within 6% in all test cases. P-Trees' throughput improves as the ratio of reads increases. This is because each read transaction on the P-Tree operates on a snapshot of the tree and is not blocked by other transactions.

For the Insert-only workload, P-Trees outperform OpenBw and MassTree, but is 10% slower than B+tree, and 20% slower than Chromatic tree. Chromatic trees' high update throughput is at the expense of low read performance (workload C). B+trees' better performance is likely to come from shallow height and better cache locality. On the other three workloads, P-Trees generally demonstrate better performance than the other data structures. The main overhead in our DBMS is the overhead of batching since our DBMS has to buffer all the transactions' operations. Meanwhile, P-Trees's good performance comes from better parallelism and non-contention achieved by batching. We will discuss more details later.

Latency vs. Throughput To better understand how batching affects the performance of P-Trees, we next measure the system's performance when varying the acceptable latency, which is done by adjusting the batch size and waiting time between batches. We control the 99% of longest time (P99 latency) each transaction waits for a response, and test the throughput of P-Trees. We then compare this against the best performance of the other data structures measured in Figure 10.2(a) for any thread count configuration. We first load 50m entries into the database and then use 144 threads to execute the YCSB Workload A transactions.

The results in Figure 10.2(b) show that, to match the best of the other four data structures, P-Trees only causes ∼10 ms latency. At a 50 ms latency window, P-Trees are more than 2.2× faster than all the other indexes. However, with more strict restriction in latency, e.g., when only 5 ms latency is allowed, the P-Tree's performance is worse than all the other tested data structures.

Scalability. Lastly, we measure how the indexes perform as we scale up the number of concurrent threads. We first observe that the results in Figure 10.3(a) show that most of the data structures are able to scale on the insert-only workload. For the mixed workloads, Figures 10.3(b) to 10.3(d) show that P-Trees achieve good scalability and parallelism, and is the only index that scales up to 144 threads in all workloads. The other four indexes suffer from bad performance with more threads in workload A (50/50 reads/updates). This is expected as more concurrent threads create more contention in the data structure. P-Trees avoid this contention by allowing reads to access an isolated tree structure. For write transactions, our implementation batches and executes them in a parallel divide-and-conquer algorithm. This also avoids contention because no two threads work on the same tree node at the same time.

Chapter 11

HTAP Database Management Systems with Snapshot Isolation

11.1 Introduction

There are two major trends in modern data processing applications that make them distinct from database applications in previous decades [230]. The first is that analytical applications now require fast interactive response time to users. The second is that they are noted for their continuously changing data sets. This poses several challenges for supporting fast and correct queries in DBMSs. Foremost is that queries need to analyze the latest obtained data as quickly as possible. Data has immense value as soon as it is created, but that value can diminish over time. Thus, it is imperative that the queries access the newest data generated, without being blocked or delayed by ongoing updates or other queries. Secondly, the DBMS must guarantee that each query has a consistent view of the database. This requires that the DBMS atomically commit transactions efficiently in a non-destructive manner (i.e., maintaining existing versions for ongoing queries). Finally, both updates and queries need to be fast, e.g., exploiting parallelism or employing specific optimizations.

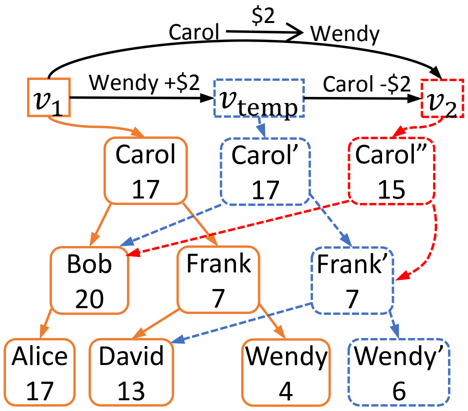
To address concurrent updates, one solution is to use multi-version concurrency control (MVCC) [62, 211, 269]. Instead of updating tuples in-place, with MVCC each write transaction creates a new version without affecting the old one so that readers accessing old versions still get correct results. In *snapshot isolation* (SI) every transaction sees only versions of tuples (the “snapshot”) that were committed at the time that it started. Many DBMSs support SI, including both disk-oriented [183, 233] and in-memory [214, 248, 273] systems. The most common approach to implement SI is to use *version chains* [60, 237, 273], which maintains for each tuple a list of all its versions. A drawback of version chains, however, is that it can make readers slower: finding a tuple that is visible to a transaction requires following pointers and checking the visibility of each tuple version. One can reduce this overhead by maintaining additional metadata (e.g., HyPer creates *version synopses* [214]) about tuples, but those approaches still have overheads.

Modern DBMSs employ several approaches to accelerate read-heavy workloads but usually at the cost of slower updates. For example, columnstores allow for better locality and parallelism, such that queries accessing the same attribute within multiple tuples run faster [85]. However, it makes insertions and deletions more expensive [116], and also require delicately-designed locking schemes that can inhibit certain updates to a tuple or version chain [61, 181, 226, 237]. Another way to improve OLAP performance is to denormalize tables or use materialized views to pre-compute intermediate results for frequently executed queries. Both of these approaches make updates more expensive because of the overhead of updating tuples in multiple locations [193] or invalidating the view.

To achieve high-performance in both updates and queries, we use P-Trees for multi-versioned, in-memory database storage. P-Trees provide three important benefits for HTAP workloads on multi-core architectures. Foremost is that P-Trees are pure (immutable, functional) data structures (i.e., no operations modify existing data). Instead of version chains, P-Trees use path-copying, also referred to as copy-on-write (CoW), to create a new "copy" of the tree upon update. This means that the index itself is the version history without requiring axillary data structures. Figure 11.1 presents an illustration of using path-copying in DBMSs for SI. Second, the trees use divide-and-conquer algorithms that parallelize bulk operations on tables—including filter, map, multi-insert, multi-delete, reduce, and range queries. These algorithms are based on using efficient operations that split and concatenate trees [74, 262], referred to as *join*-based algorithms, and an efficient work-stealing scheduler for fork-join parallelism [2, 84]. Lastly, P-Trees support the *nesting* indexes inside of themselves. Such nested indexes improve OLAP query performance while still allowing for efficient updates under SI.

Using a pure data structure means that the DBMS must serialize updates to the global view or combine them together into batches. Some previous CoW-based systems, like LMDB [4], only allow for a single active writer and thus serialize writes. Others, like Hyder [63, 64, 238], support "melding" trees, but the melding process is still sequentialized, and furthermore can cause aborts. These limitations are the major bottleneck in CoW-based systems. P-Trees exploit parallelism by supporting parallel bulk operations as mentioned in Chapter 10. For any large transactions, or batched transactions consisting of multiple insertions and deletions, P-Trees can leverage multiple cores to atomically commit database modifications in parallel.

To evaluate P-Trees, we compare them with state-of-the-art concurrent data structures and in-memory DBMSs (MemSQL [248] and HyPer [214]). Our results show that on the OLTP workload YCSB (consisting of searches and updates) P-Trees are competitive with existing concurrent data structures that do not support multi-versioning. On an OLAP workload TPC-H, P-Trees are 4-9× faster than MemSQL and HyPer on a 72-core machine.



□ **Committing** v_2 as the latest version only requires to swing in v_2's root.

□ **Rolling back** can be done by re-attaching the current version to v_1's root.

□ The changes in the transaction (Carol -$2, Wendy +$2) are visible **atomically**.

□ v_{temp} can be **collected**. Only the node Carol' will be freed because the other nodes are referenced by other trees.

□ A transaction **reads** a version by acquiring the root pointer. An **update** transaction generate a new version by copying nodes on the affected path. A **read** or an **update** on a single tuple costs time $O(\log n)$.

Figure 11.1: P-Trees Multi-Versioning Overview – An example of using P-Trees to support a bank balance DBMS. The original state is v_1. A transaction transfers $2 from Carol to Wendy.

On an HTAP benchmark with ongoing updates, P-Trees remain almost as fast on the queries while supporting update rates comparable to what is supported by MemSQL (and much faster than HyPer). All the systems, including ours, support durability. Our queries are coded directly rather than using the SQL code, and we therefore exclude compilation time for MemSQL and HyPer. We also study what contributes to our performance gain of P-Trees compared to the other systems on queries, by removing some of the features. Our results show that much of the improvement is due to better parallel scaling (62× speedup on average using 72 cores) and the index nesting optimization (2× performance improvement on average).

Our contributions can be summarized as follows.

1. The combination of path copying and parallel bulk operations in an MVCC database is new. The bulk parallelism leads to very good speedup and performance on both queries and batch updates while supporting full serializability.
2. The use of nested indexes based on trees with their application to fast queries. This leads to significant speedup on many of the TPC-H queries.
3. The C++ implementation of a DBMS based on P-Trees. This includes the implementation of nested indexes, and support for parallel bulk operations. It also includes the implementation of all 22 of the TPC-H queries. We have made the code available on GitHub [8].
4. An experimental evaluation comparing our code to other systems, and analyzing the effectiveness of various features of our system.
5. A new benchmark that adds TPC-C style transactions to TPC-H. This differs from previous CH-benCHmark [113] on adding TPC-H style queries to TPC-C.

We note that although some of the contributions seem independent (e.g. bulk operations and nested indexes), an important aspect of the work is that P-Trees make it particularly easy to combine these ideas.

11.2 Approach

In this section, we discuss how parallel bulk algorithms for updates and queries can be used on database queries. All algorithms in this section are pure using path-copying.

P-Trees can be used to maintain a sorted set of *key-value* pairs. We allow the key and value to be of any type, such as integers, strings, or even another tree. As we will discuss in Section 11.3, supporting nested tree structures is beneficial for OLAP workloads. In our implementation, P-Tree is the sole date representation. In our experiments, all dynamic data can only be accessed through the P-Trees.

11.2.1 Parallel Bulk Update Operations

We use *multi_insert* and *multi_delete* to commit a batch of write operations. The function *multi_insert*(t, A, m, σ) takes as input a P-Tree root t, the head pointer of an array A with its length m, and a combine function σ. The combine function $\sigma : V \times V \mapsto V$, where V is the value type of the tree is used to deal with duplicate keys. The algorithm is introduced in Section 3.3.4. When inserting a key-value pair $\langle k, v\rangle$, if k is already in t, then the tree updates its value by combining the value of k in t with v using σ. This is useful, for example, when the value is the accumulated weight of the key and the combine function is addition ($\sigma(a, b) = a + b$). Another use example is when installing new values for existing keys ($\sigma(a, b) = b$). The technique is introduced in Section 3.3.5.

A DBMS uses these parallel bulk update algorithms to commit a batch of operations to the database. We will discuss more details in Section 11.4. For concurrent point updates, we combine our approach with the batching algorithm in 10.2.1.

11.2.2 Parallel Bulk Analytical Operations

To facilitate read-only analytical queries, P-Trees support several analytical primitives, including *range*, *filter*, *foreach_index* and *map_reduce*. The algorithms are introduced in Section 3.3.4. Some of these algorithms return trees as output. This is useful for maintaining intermediate results of primitives in the same representation as the input (i.e., another index structure) since this allows primitive cascading; this is also known as *index spooling* in Microsoft SQL Server [3]. The P-Trees primitives extract such intermediate views on the current snapshot and output another tree structure. This not only avoids additional data scans, but is also asymptotically more efficient than scanning the data directly. For example, although intuitively extracting a range would take time proportional to the range size, our method avoids doing so by outputting a *tree*, and thus avoids touching output data—only the affected path is read and copied, instead of the whole range of entries. This is useful, for example, in cascading queries when the output is for further queries.

11.3 Nested Indexes

To support efficient analytical queries, the P-Tree can use *nested* and *paired* indexes. A nested index embeds one index inside another such that each value of the top level index is itself another index. A paired index uses a single index for two tables that share the key, often a primary key in one and a secondary or foreign in the other. We show an example in Figure 11.2 on the TPC-H workload, and will explain it in detail later in this section. Both nested and paired indexes can be considered a "virtual" denormalization of the data. In particular, a paired index is logically a pre-join on the shared key, and a nested index roughly corresponds to adding a pointer from the parent to the child and indexing on it (sometimes referred to as the *short-circuit key*). However the nesting and pairing does not materialize the view—there is no copying of the tuples, and therefore it suffers less from the standard problems of additional space, consistency, and expensive updates. We will discuss the space overhead of index nesting in Section 11.4.

Nesting and pairing is straightforward with P-Trees since it supports arbitrary key and value types. Furthermore, nested and paired tables work well with both its parallel operations and with path copying. These operations over a nested index, such as map and reduce, can themselves be nested so there is parallelism both on the outer index and inner index (we provide an example below). Path copying works with nesting since the path from an outer tree continues into a path on the inner nested tree (see Section 11.4).

In general, one can apply nesting across multiple levels. For example, for TPC-H we create up to three levels (e.g., customer-order-lineitem, or part-supplier-lineitem). We also use nested indexes for indexes on secondary keys—the outer index is on the secondary key and each inner index contains the tuples with the same secondary key indexed by their primary key. This differs from the more common implementation of secondary indexes that keeps inner sets of elements with the same secondary keys as lists or arrays [142]. Using a nested index has the advantage of being able to quickly find and delete an element in the inner index based on its primary key.

We are not aware of any DBMS that uses index nesting based on trees. This is possibly because such nesting is unlikely to be efficient on disk-based DBMSs due to fine grained memory accesses, and because it is complicated to to combine nesting with existing optimization techniques, such as column stores or version chaining. P-Trees make such nesting easy to support, and we do use a columnar data model or version chaining.

To illustrate how nesting and pairing works, we consider an example from TPC-H shown in Figure 11.2. The top level index for the `ORDER` table is on the non-unique `orderdate` key (i.e., the date on which a customer made an order). Within each date, there is an index of the orders on that date keyed on the primary key `orderkey`. In the

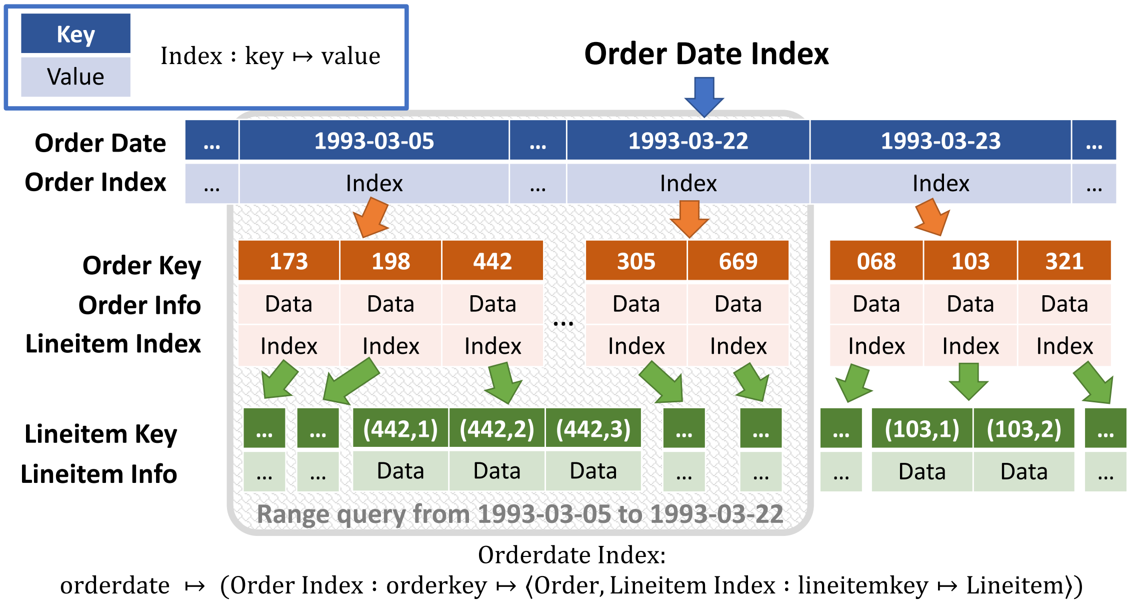


Figure 11.2: Nested Tree Structure in Our TPC-H Implementation – The orderdate-order-lineitem relation in TPC-H. A range query on the top level index would effectively filter out irrelevant lineitems in the bottom level, making queries more efficient.

TPC-H schema, each order has a set of items called lineitems, or more precisely, each lineitem has an `orderkey` as part of its two-attribute primary key. Since lineitems share the `orderkey` with orders, we can pair the indexes. This pairing is shown by the two orange tuples for "Order Info" and "Lineitem Index". The order info contains the complete table entry for each order. This is either (1) the tuple directly stored in the tree node or (2) a pointer to the tuple elsewhere in memory. In our experiments, we compare the two methods. Importantly, the example shows a third level of nesting on the lineitems themselves, indexed based on their primary keys, which consists of both the `orderkey` and the `linenumber`. Each of the inner most indexes is itself represented as a P-Tree, even though in TPC-H there are at most seven lineitems per order. The example shows both three-level nesting and pairing.

This nested index can be thought of as a virtual pre-join of the `ORDER` and `LINEITEM` tables on their shared `orderkey`, and then indexing the result based on `orderdate`. As the illustration shows, however, the orders are not copied across multiple lineitem tuples, and the `join` is never materialized. We note the same index can be used to answer range queries by date on just orders, or on the lineitems that belong to a range of `orderdates`.

To show why the nesting and pairing are useful, we consider TPC-H Q4. We consider both how existing DBMSs process the query and then how the nested-paired index can be used to significantly improve performance. The query in SQL is given in Figure 11.3(a).

```
SELECT o_orderpriority, COUNT(*) AS order_count FROM orders
 WHERE o_orderdate ⟩= date '[DATE]'
   AND o_orderdate ⟨ date '[DATE]'+interval '3' month
   AND EXISTS ( SELECT * FROM lineitem
                WHERE l_orderkey = o_orderkey AND l_commitdate ⟨ l_receiptdate )
 GROUP BY o_orderpriority
 ORDER BY o_orderpriority;
```

(a) SQL

```
using Arr = Array⟨int, NUM_PRI⟩; //NUM_PRI is 5
Arr Q4(DB d, const Date date) {
  orderdate_tree t = d.orderdate_idx.range(date, date.add_month(3));
  auto date_map_func = [] (orderdate_entry& odate_entry) -⟩ Arr {
    auto order_map_func = [] (order_entry& order_entry) -⟩ Arr {
      auto item_map_func = [] (Lineitem& l) -⟩ bool {
        return l.commit_date⟨l.receipt_date;};
      int pri = order_entry.get_order().orderpriority()-1;
      Arr a;
      a[pri]=order_entry.get_item_idx().map_reduce(item_map_func, OR());
      return a; };
    return dt.get_order_idx().map_reduce(order_map_func, Add_Array()); };
  return t.map_reduce(date_map_func, Add_Array()); }
```

(b) P-Tree Implementation

Figure 11.3: TPC-H Q4 – The definition TPC-H Q4 and the pseudocode of using map-reduce functions on P-Trees for implementing TPC-H Q4. *OR* in the code means the logical-OR operation on boolean values.

The query looks up the orders where (1) the orderdate is in a given range and (2) there exists a lineitem for the order such that `commitdate < receiptdate`. It then counts the number of relevant orders for each different `orderpriority` (i.e., five). This query accesses both the `ORDER` and `LINEITEM` tables. But DBMSs often scan the `LINEITEM` table first, which is problematic because there are 4× tuples as the the total number of orders, and 28× as many as the number of orders in the `orderdate` range. We discuss this problem further in Sections 11.6.2 and 11.6.3 for MemSQL and HyPer.

We now consider how to perform the query using the nested index illustrated in Figure 11.2. The DBMS can first do a range search on a specific date range, identifying the ⟨order, lineitem index⟩ pairs that fall within the range (i.e., the orders in the shaded rectangle in Figure 11.2). For TPC-H Q4, this is about 1/28 of all the orders. Then for each such order it can examine all the lineitems that belong to the order to see if any satisfy the predicate on order and commit date. If so, then the system increments a counter for the appropriate order priority, which it can combine with a parallel reduce.

Our code for Q4 using P-Trees is given in Figure 11.3(b). It extracts the range of order dates in Line 3. It then has nested parallel calls to `date_f` (over `orderdates`), then `ord_f` (over all orders for a given date), and finally `item_f` (over all lineitems of a given order). The outer two both use `map_reduce` and the inner-most just checks if the lineitem satisfies `commit_date` < `receipt_date`. This producs an approximately 10 fold improvement over MemSQL, HyPer, and our own DBMS without nested indexes (see Section 11.6.2). We also added the secondary index on `orderdate` in HyPer, but it did not help since the system still needed to scan the `LINEITEM` table.

The general idea of index nesting does not rely on TPC-H or the P-Tree, and thus is of independent interest and can be extended to other settings.

11.3.1 Defining Nested Indexes

In our DBMS, we supply a simple way for users to construct nested indexes by building them up based on three primitives. All the indexes, both the outer and inner ones, are maintained by P-Trees. To construct a nested index, we define the following functions.

1. `primary(Table, primarykey)`: constructs an index from a table `Table` based on the `primarykey`.
2. `secondary(Index, secondarykey)`: constructs a secondary index from a primary index `Index`, based on the `secondarykey`. If there are multiple tuples sharing the same secondary key, build inner indexes based on the primary key.
3. `pairing(Index1, Index2)`: for two indexes `Index1`: $X \mapsto Y$ and `Index2`: $X \mapsto Z$, construct an index that maps X to a pair of Y and Z. This is similar to a regular *join* operation, but keep the join column as the key of the output index.

These three primitives fully support the index nesting we propose in this chapter. We have implemented the three functions based on P-Trees and use them in our experiments.

We continue our example of the index from Figure 11.2:

```
      I0 = primary(LINEITEM, lineitemkey)
      I1 = primary(ORDER, orderkey)
      I2 = secondary(I0, orderkey)
   I_ORD = pairing(I1, I2)
 I_ODATE = secondary(I_ORD, orderdate)
```

The first two lines build primary indexes on lineitems and orders, the third line then builds a secondary index for the lineitems based on the `orderkey`, and the fourth line pairs up the primary order index `I2` with this secondary lineitem index `I3`. The last line indexes the paired index based on the secondary key `orderdate`, and returns a three-level index `I_ORD`.

As another example of building a three level index, the following will construct a customer index on top of the `I_ORD` index.

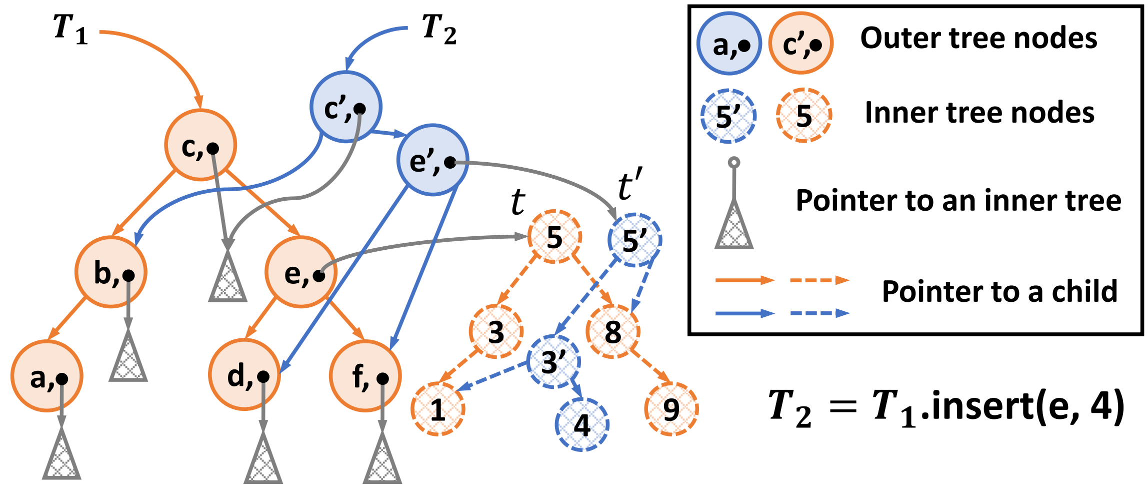


Figure 11.4: Functional Update on a Multi-level P-Trees – The update uses path-copying on both inner and outer trees.

```
          I3 = primary(CUSTOMER, custkey)
          I4 = secondary(I_ORD, custkey)
      I_CUST = pairing(I3, I4)
```

It is a three-level nested index, each level also is a paired index with customer and order tuples stored in it, respectively. This index is used frequently in our implementation of TPC-H queries. In queries that require a *join* on lineitems and a selection of customers (e.g., Q3), we can filter out the irrelevant customers on the top level such that their lineitems will not be scanned. In TPC-H, there are 40× more lineitems than customers, so the total accessed data will be much less than all the lineitems. The improvement of using a nested index in Q3 is more than 300% (see Table 11.3).

11.4 Using Pure P-Trees for SI

In this section, we describe how to use pure P-Trees for SI and MVCC, and how to achieve serialibility using batching. P-Trees use pure *join*-based algorithms that never modify existing tree nodes, but copy necessary parts when updates occur. As a result, multiple logical versions of indexes share physical tree nodes.

MVCC Example. We use the example in Figure 11.1 to show how pure P-Trees support MVCC and SI. In P-Trees, a transaction acquires the current version of an index by grabbing the root and incrementing the root's reference counter to create a snapshot of the index. When our system must maintain multiple consistent indexes, it keeps a top-level tuple storing a pointer to each index. We refer to this as the "world" since it stores all dynamic information for a database. Acquiring a version grabs a pointer to this tuple. Creating a new world on updating an index requires copying this tuple and putting in the new root of the updated index. The tuple (the world) is only a constant number of pointers.

The DBMS creates local versions of indexes using analytical operations, such as *range* and *filter*, but it does not need to commit these local versions. Any updates also create new versions, but the DBMS commits them by writing a pointer to the new world. Figure 11.1 shows an example of a transaction on a single index. The transaction transferring \$2 from Carol to Wendy is implemented by two consecutive updates using path-copying, leading to a new version v_2. Committing the new version v_2 involves updating the current root pointer to v_2's root. Then the two updates in the transaction become visible atomically. Such atomic updates are applicable to either a series of updates like in this example, or a batch of updates (e.g., by a pure *multi_insert*). The old versions are still available and can be accessible by the old root pointers, so ongoing queries can continue working on them. Such a strategy also makes rollbacks easy by swinging back the old root pointer. The DBMS's GC algorithm can collect old versions.

Using P-Trees for MVCC requires no additional versioning or other internal auxiliary fields in the data structure. The DBMS can maintain versioning information (e.g., timestamps and liveness), with the set of root pointers to the versions.

Pure Update on Nested Trees. As described in Section 11.3, P-Tree's index nesting accelerates the analytical queries. Using P-Trees to store nested indexes has the benefit that updates are inexpensive, as the DBMS can perform the update by path-copying across both the outer and inner trees. Figure 11.4 shows an illustration of updating a two-level P-Tree. The update algorithm first finds the affected tree node *e* in the outer tree and then copies the path along the way. For the other copied nodes in the outer tree other than $\boxed{e}$, the inner trees do not change, and thus we can directly use the pointer to the original inner trees. For example, node $\boxed{c'}$ in T_2 has the same inner tree pointer as $\boxed{\mathrm{c}}$ in T_1. For $\boxed{e}$ itself, we copy it to $\boxed{e'}$, and the algorithm then inserts $\boxed{4}$ to the inner tree t of $\boxed{e}$ (the orange slashed nodes), giving a new inner tree t'. The root of t' is then assigned to $\boxed{e'}$ as the inner tree. The total cost of such an insertion is $O(\log n_I + \log n_O)$, where n_I and n_O are the sizes of the inner and outer trees respectively. As such, SI is still supported, and all algorithms in Section 11.2 remain applicable on a nested tree, or on multiple nested trees stored in a world.

11.4.1 Serializable Updates

One concern with SI is in supporting serializability for concurrent update transactions [97]. This is further complicated in path-copying-based (pure) approaches since each *update* makes its own copies of paths, which then need to be resolved if they run concurrently. A simple solution is to only allow a single writer to sequentialize all *update*s (e.g., flat-combining [155]). This is likely to be adequate if *update*s are large with significant internal parallelism, or if the rate of smaller *update*s is light (in the order of tens of thousands per second), as might be the case for a DBMS dominated by analytical queries. It is unlikely to be adequate for DBMSs with high update rates of small transactions. Hyder [238], which uses multiversioning with path copying, addresses this by allowing transactions to proceed concurrently and then merging the copied trees using a “meld” operation. The DBMS, however, must sequentialize these melding steps so that it creates new versions one at a time. The melding process can also cause an abort even when the updates are logically independent—for example when the first update does a rotation on an internal tree node that the second visits.

Another approach is to batch updates as part of a group commit operation [121]. The basic approach is for the DBMS to process a set of updates obeying a linear order. It then detects any logical conflicts based on this order and the operations they performed (e.g., write-read conflicts). The system next removes any conflicted updates and then commits the remaining conflict-free updates as a batch. This approach is taken by a variety of systems, including Calvin [266], FaunaDB [1], PALM [247], and BATCHER [18]. The advantage of this approach is that the batch update can make use of parallelism [1, 18], consensus in only needed at the granularity of batches in distributed systems [1, 266], and the batched updates can make more efficient use of the disk in disk-based systems [30]. The challenge of the approach, however, is in detecting conflicts; all of the above DBMSs perform this step differently.

In our system we use batching and use multi-insert and multi-delete to apply batches of point updates. As discussed, these primitives have significant internal parallelism. In this chapter we do not consider how to detect conflicts for arbitrary transactions (previous work could help here), but study the approach for simple point transactions such as in the YCSB benchmark (insertions and deletions). In this case, the only conflicts are between updates on the same key, and keeping the last update on the key is sufficient. We note it is also easy to detect conflicts in the TPC-C transactions we evaluate: for the New-Order transactions described in Section 11.6.1, one can check if two transactions share a customer or part-supplier. In our experiments, our DBMS simply sequentializes them because the workload is comprised of mostly analytical queries.

To batch a set of updates, we wait for some amount of time, allowing any updates to accumulate in a buffer, and then apply all the updates in the buffer together as a batch in parallel. While processing the batch new updates can accumulate in another buffer. In this approach there is a tradeoff between throughput and latency—the longer we wait (higher latency), the better the throughput due to increased parallelism. This tradeoff is analyzed in Section 10.3.2.

11.5 Space Overhead

Here we discuss the space overheads of our approach.

Space Overhead from Index Nesting. P-Tree’s nesting method is similar to data denormalization and materialized views, but avoids copying of data across rows. This both saves memory and reduces the cost and complexity of updates. However, there is some space cost of index nesting. In particular, each nested index in which a table row can appear can cause a copy of that row. In our TPC-H implementation, for example, the lineitems are copied four times in different nested indices because each lineitem is involved in several hierarchies and secondary indexes. The copy can either involve storing the row directly in a tree node, or a pointer from the tree node to a shared copy of the row. The second approach requires less memory since the row is shared among indices only requiring a pointer within each index. However, it might require extra time in analytical queries due to a level of indirection, and the additional cache misses this incurs. One can also save space by using fewer secondary indexes, which also slows down some queries. In our experiments, we compare these approaches and analyze the tradeoff.

Space Overhead from Path-copying. Using path-copying inherently copies more data than other MVCC solutions based on version chains since it copies the whole path to the update rather than just the affected tuple itself. However, it needs less metadata within the data structures (e.g. timestamps on each version within a version list). It also can easily garbage collect any old versions, which is complicated with version chains. To be more concrete, to insert or delete a batch of m tuples into a tree of size $n \geq m$, the number of extra tree nodes created by our *multi_insert* algorithm is $O\big(m(\log \frac{n}{m} + 1)\big)$ [74]. This may seem high, but it is always asymptotically bounded by n (the size of the original index). In practice it is usually small since m is much less than n. However, if many versions are kept, this cost can accumulate over the versions if they are not collected. In section 11.6.4 we experimentally evaluate this memory overhead as a function of the number of versions.

11.6 Experimental Evaluation

We now provide a comprehensive evaluation of P-Trees under a variety of settings and workload conditions. For all of these experiments, we use a 72-core Dell R930 with four Intel Xeon E7-8867v4 (18 cores, 2.4GHz, and 45 MB L3 cache), and 1 TB memory. Each core is two-way hyperthreaded giving a total of 144 hyperthreads. Our code was compiled using g++ 5.4.1. We use `numactl` in the experiments with more than one thread to spread the memory pages across CPUs in a round-robin fashion.

We first compare P-Trees with four concurrent data structures on OLTP workloads. We then compare our system to two DBMSs HyPer [178] and MemSQL [248] on both an OLAP and HTAP workloads. Finally we analyze the space overhead of our system from path-copying and index nesting.

11.6.1 Workloads

We first describe the benchmarks we use in our evaluation. For our P-Tree, we implement the benchmarks in our testbed DBMS. For the SQL-based systems (HyPer and MemSQL), we use the open-source OLTP-Bench benchmarking framework [124].

TPC-H. We use this OLAP workload to evaluate the performance of our DBMS executing analytical queries. Of the eight tables in the TPC-H database, we use seven nested

P-Tree structures (labeled with) to maintain primary and secondary indexes. The first two indexes defined below are used only as inner trees. The exact configuration of the database is formalized as follows:

$$\begin{aligned}
&\textbf{Inner indexes:} && \\
&T_{\text{lineitem}} = \mathbb{T} : \langle \text{orderkey, linenumber} \rangle \mapsto \text{lineitem} && \\
&T_{\text{partsupp}} = \mathbb{T} : \langle \text{partkey, suppkey} \rangle \mapsto \langle \text{partsupp}, T_{\text{lineitem}} \rangle && \\
&\textbf{Primary indexes:} && \\
&T_{\text{order}} = \mathbb{T} : \text{orderkey} \mapsto \langle \text{order}, T_{\text{lineitem}} \rangle && (*) \\
&T_{\text{cust}} = \mathbb{T} : \text{custkey} \mapsto \langle \text{customer}, T_{\text{order}} \rangle && (*) \\
&T_{\text{supp}} = \mathbb{T} : \text{suppkey} \mapsto \langle \text{supplier}, T_{\text{partsupp}} \rangle && (*) \\
&T_{\text{part}} = \mathbb{T} : \text{partkey} \mapsto \langle \text{part}, T_{\text{partsupp}} \rangle && (*) \\
&\textbf{Secondary indexes:} && \\
&T_{\text{receiptdate}} = \mathbb{T} : \text{date} \mapsto T_{\text{lineitem}} && (*) \\
&T_{\text{orderdate}} = \mathbb{T} : \text{date} \mapsto T_{\text{order}} && (*) \\
&T_{\text{shipdate}} = \mathbb{T} : \text{date} \mapsto T_{\text{lineitem}} && (*)
\end{aligned}$$

$\mathbb{T} : K \mapsto V$ denotes a tree storing a mapping from K to V. $\langle A, B \rangle$ means a pair of two elements A and B. The value type can also be an index represented by another tree. We keep four arrays for static data on the SUPPLIER, PART, NATION, and REGION that map the primary key to the corresponding tuple. We note that none of the nested indexes created in our DBMS is just for a specific query. For example, a total of six different TPC-H queries take advantage of the three-level customer index T_{cust}. We optimize the layout for read-heavy workloads, and thus store all tuples directly in the trees. This may bring up extra cost in updates. For example, when updating an order, our DBMS modifies three indexes.

We report the geometric mean of the running time for each query across five trials, and also report the geometric mean of all 22 queries, as is suggested in the TPC-H official document [7].

TPC-HC. To provide a more in-depth analysis of hybrid workloads, we created a hybrid benchmark called TPC-HC based on TPC-H and TPC-C. Unlike the HyPer's CH-benCHmark [112], which integrates TPC-H queries into TPC-C, our benchmark integrates TPC-C transactions into TPC-H workloads. We do this because our DBMS is optimized for OLAP queries, and thus we want to provide a more fair comparison with other systems on TPC-H queries.

This benchmark contains all 22 queries from TPC-H along with the following three transactions derived from TPC-C with their denoted percentage of the total update workload:

1. The **New-Order [49%]** transaction, which is derived from the TPC-C *New-Order* transaction. In this transaction, a new order is committed by a random customer. It contains a random number of lineitems, each randomly selected from all the part-suppliers. In particular, a new order is generated as follows:
   (a) The O_ORDERDATE is set to the current timestamp. This new order has a random O_SHIPPRIORITY (1-5), random O_ORDERPRIORITY (1-5). The O_ORDERSTATUS is set to ‘O’. The order id is the current maximum order id increased by 1.
   (b) A set of x ($1 \leq x \leq 7$) lineitems are in this order. For the i-th lineitem:
      - It is uniformly randomly chosen from all part-suppliers.
      - The L_QUANTITY is a random integer in [1, 50].
      - L_EXTENDEDPRICE = L_QUANTITY × P_RETAILPRICE, where the P_RETAILPRICE is extract from the PART table where P_PARTKEY = L_PARTKEY.
      - L_LINEITEM = i. L_LINESTATUS = ‘O’. L_INSTRUCTIONS = “NONE”. L_SHIPMODE = “RAIL”. L_RETURNFLAG = ‘R’.
      - L_SHIPDATE, L_COMMITDATE, L_RECEIPTDATA are all set to an infinity date.
      - All the others columns of this lineitem are set to the default value or empty.

   (c) The O_TOTALPRICE computed as: sum (L_EXTENDEDPRICE × (1 + L_TAX) × (1 - L_DISCOUNT)) for all lineitems of this order.
   (d) All the others columns of this order are set to the default value or empty.

   It requires the following changes to the database:
   (a) The order is added to the ORDER table, and all lineitems are added to the LINEITEM table.
   (b) The available quantity of each corresponding part-supplier decreases by 1.
   (c) Enqueue the order id to a queue Q_norder.
2. The **Payment [47%]** transaction, which is derived from the TPC-C *Payment* transaction. In this transaction, a random customer will pay a certain amount of money, and thus the customer’s balance increases accordingly. The amount of money is selected randomly in [1, 5000].
3. The **Delivery [4%]** transaction, which is derived from the TPC-C *Delivery* transaction. In this transaction, the lineitems in the earliest y orders in Q_{norder} are shipped. Also at this point, the customer will be charged the O_TOTALPRICE of money of the corresponding order. It requires the following changes to the database:

Txn	Table	Operations
New-Order	T_{cust},T_{order} $T_{\text{orderdate}}$, T_{supp}, T_{part}, Q_{order}	* Add to the following relations: partsupp ↦ lineitem, customer ↦ order, order ↦ lineitem, orderdate ↦ order * Decrease each item's available quantity * Enqueue this order to Q_{order}
Payment	T_{cust}	* Decrease the customer's balance
Delivery	Q_{order}, T_{shipdate} T_{part}, T_{supp} T_{cust}, T_{order} $T_{\text{orderdate}}$,	* Dequeue some orders from Q_{order} * Update lineitems' shipdate, customers' balance, lineitems' and orders' status, * Add to the shipdate ↦ lineitem relation

Table 11.1: Updates on Trees in TPC-HC – The trees that require updates in each transaction type.

(a) The L_SHIPDATE of the lineitems in these orders are updated to the current timestamp.

(b) Each customer of the shipped orders have their balance decreased by the value of O_TOTALPRICE.

(c) The L_LINESTATUS of each lineitem and the O_ORDERSTATUS of the corresponding orders are changed to "F".

The order and lineitem information are generated based on TPC-H specification. For our implementation, the required actions of each transaction on our TPC-H configuration is shown in Table 11.1. During each trial, the system uses one thread for the update transactions and another thread to invoke TPC-H queries. We use the same code as in our TPC-H experiments and run queries in parallel with all available threads. All of the OLTP transactions and OLAP queries operate on the latest snapshot available to them. After each OLTP transaction finishes, the DBMS commits their updates atomically. They are then immediately visible to the next TPC-H query.

All updates for P-Tree are running sequentially. We allow to run multiple update transactions at the same time for MemSQL.

Update (in an infinite loop)	**Query** (in an infinite loop)
`//read the current snapshot`	`for (int i = 1; i < 22; i++) {`
`DB d0 = *d;`	`  //read the current snapshot`
`Txn t = next_txn();`	`  DB d0 = *d;`
`DB dn = update(d0,t);`	`  query(d0, i);`
`d = &dn;`	`}`

Here d is the global variable pointing to the current version of the database. We run both queries and updates in infinite loops until some stop condition is reached.

	MemSQL					Hyper					P-Tree				
	Par. (ms)			Seq. (s)	spd-	Par. (ms)			Seq. (s)	spd-	Par. (ms)			Seq. (s)	spd-
Update	✗	✓	ratio	✗	up	✗	✓	ratio	✗	up	✗	✓	ratio	✗	up
Q1	375	436	16.3%	11.6	31.0	354	345	-2.5%	12.4	35.0	324	331	2.0%	6.6	20.5
Q2	233	295	26.6%	9.1	38.9	255	256	0.4%	9.1	35.7	15	16	9.5%	0.8	51.9
Q3	2377	2494	4.9%	12.2	5.1	441	469	6.3%	14.1	31.9	144	147	1.9%	9.4	65.6
Q4	403	504	25.1%	27.9	69.2	357	386	8.1%	14.6	40.9	36	37	3.6%	3.1	87.9
Q5	1171	1174	0.3%	36.3	31.0	507	543	7.1%	17.7	34.9	68	70	1.7%	5.0	72.4
Q6	230	310	34.8%	7.3	31.7	100	103	3.0%	1.1	11.2	51	52	0.4%	2.7	53.6
Q7	579	904	56.1%	13.2	22.8	381	393	3.1%	12.1	31.8	60	62	3.5%	4.4	72.6
Q8	298	335	12.4%	15.1	50.8	125	137	9.6%	4.8	38.6	94	96	2.7%	5.8	61.9
Q9	1726	1915	11.0%	127.9	74.1	1176	1200	2.0%	47.8	40.7	184	184	0.1%	7.9	42.8
Q10	700	808	15.4%	66.9	95.6	404	385	-4.7%	11.4	28.2	53	55	2.4%	4.2	78.4
Q11	120	124	3.3%	1.9	15.7	67	81	20.9%	1.4	20.7	14	14	2.6%	0.7	53.4
Q12	277	378	36.5%	10.2	36.9	120	121	0.8%	4.6	38.2	105	106	1.0%	10.2	97.2
Q13	4561	4033	-11.6%	279.5	61.3	1559	1645	5.5%	73.4	47.1	406	414	2.0%	34.3	84.4
Q14	250	244	-2.4%	14.7	58.8	79	249	215.2%	7.0	88.5	22	25	13.3%	1.6	71.0
Q15	1131	1795	58.7%	26.4	23.3	204	221	8.3%	3.0	14.7	25	27	7.9%	1.3	49.6
Q16	660	730	10.6%	35.8	54.2	426	436	2.3%	7.2	16.8	70	73	3.6%	5.7	81.1
Q17	258	265	2.7%	8.1	31.3	261	285	9.2%	7.2	27.7	36	37	5.2%	1.2	34.9
Q18	6327	6762	6.9%	43.1	6.8	3135	3484	11.1%	37.7	12.0	425	428	0.8%	29.3	68.9
Q19	180	208	15.6%	10.9	60.4	222	240	8.1%	8.5	38.3	28	30	5.7%	2.2	77.4
Q20	1737	2012	15.8%	95.0	54.7	192	335	74.5%	9.0	47.0	25	26	2.7%	2.3	90.6
Q21	1333	1402	5.2%	363.2	272.5	798	858	7.5%	30.5	38.2	263	276	5.1%	15.5	59.1
Q22	613	640	4.4%	26.5	43.2	181	187	3.3%	6.1	33.9	39	42	8.8%	2.5	65.0
Mean	640.5	734.3	14.6%	24.7	38.5	311.3	352.7	13.3%	9.6	30.8	66.4	68.9	3.9%	4.1	62.2

Table 11.2: TPC-H Measurements (SF 100) – "seq." means sequential running time in seconds. "par" means parallel running time in milliseconds. ✓ or ✗ means with or without updates running at the same time. Gmean=geometric mean across five runs. Running time in the last row shows the geometric mean of the each column. The mean of ratios are calculated from the mean of the running time. For parallel runs, we use all 144 threads. We highlight (the grey cells) the highest throughput numbers among the three implementations.

11.6.2 OLAP Workload Evaluation

For this next group of experiments, we test P-Trees on all 22 queries in TPC-H and compare it with two in-memory DBMSs:

- **MemSQL (v6.7):** We load all of the TPC-H tables into the DBMS's column-store engine. We then use the `memsql-optimize` tool to configure the DBMS's runtime components. Although we ran our experiments on a single node, MemSQL does not allow one to create a database with a single partition, where a partition is MemSQL's granular unit of query parallelism. Thus, we use the `memsql-optimize` tool to configure the DBMS to create a two partitions per CPU.
- **HyPer (v20181.18):** We use the commercial version of the HyPer binary shipped in the Tableau distribution. We use DBMS's default runtime configuration that sets inter-query and intra-query parallelism to use all available cores. For our restricted run measurements (i.e., *sequential time*), we limited HyPer's inter-query and intra-query parallelism to one core each.

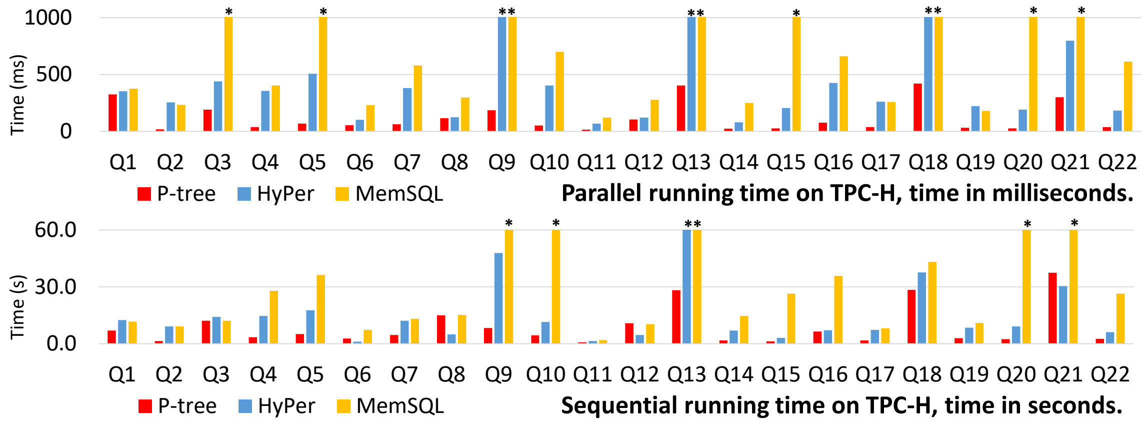


Figure 11.5: Parallel TPC-H Running Time with SF 100 – The parallel (144 threads, time in milliseconds) running time of TPC-H queries. We cut of the y-axis at one second. "*" in the figure indicates that those numbers exceed the y-axis.

For both MemSQL and HyPer there are many settings and we made a significant effort to run the workload in the best possible way. This included contacting the developers of both and using the configurations suggested by them for the TPC-H workload. The numbers we report are comparable to those which are self reported for the two database system [104, 188] on TPC-H and adjusted for processor capabilities.

We note that MemSQL's column-store engine does not support additional indexes. For HyPer we ran with and without secondary indexes and took the best time. Our implementation with P-Tree uses secondary indexes on `orderdate`, `shipdate`, and `receiptdate`. P-Trees do not use optimizations based on column stores (everything is a row store), NUMA optimizations, or vectorization.

We note that both HyPer and MemSQL are full-functional DBMS, which have more functionalities than our DBMS, including the compilation of SQL, durability, etc. To make a more fair comparison, we exclude the query compilation time for them, and also make our DBMS durable by writing logs of transactions to disk.

We use TPC-H scale factor (SF) 100 and run all tests five times to achieve more stable results. The results are presented in Table 11.2 in the columns labeled with "✗" in updates. The columns labeled with "seq." show the sequential running time, and the columns with "par." are parallel running time. We use grey cells to note the best throughput number among the three systems.

Parallel Performance. All three systems achieve stable query performance across runs. The geometric standard deviation of P-Trees, HyPer, and MemSQL are 1.015, 1.023 and 1.037, respectively. In all 22 queries, P-Trees are faster than both HyPer and MemSQL. P-Trees are at least 3× faster on most queries compared to both MemSQL and HyPer. However, in some queries (e.g., Q1, Q6, Q8, Q12 and Q21) that P-Trees cannot take much advantage of using nested indexes, P-Trees' advantage is less significant (only 1-2× faster). For example, Q1 and Q6 requires to scan several specific columns of almost all lineitems, and thus nested indexes cannot help to pre-filter. Q8 involves two nestings (customer-order-lineitem and part-partsupp-lineitem), and thus using hierarchical map-reduce function on one index still need an extra lookup at the parent-child relation in the other index. Q12 and Q21 are similar. On the other hand, in these queries an implementation could benefit from using columnstore by reducing I/O, as both HyPer and MemSQL do. In fact, the good performance of P-Trees mainly comes from two aspects: better parallelism, and the index nesting. We will discuss this in more detail later in this section.

The geometric mean of all 22 queries using P-Trees is 4.2× faster than HyPer and 9.2× faster than MemSQL.

Sequential Performance and Scalability. Overall, the P-Tree's performance is still better than HyPer and MemSQL, but not as significant as for parallel performance. For the geometric mean of the sequential running time of all 22 queries, the P-Tree is about 2× faster than HyPer and 5× faster than MemSQL. Similarly, in the queries (e.g., Q6, Q8, Q12 and Q21) that P-Trees cannot take much advantage of nested indexes, P-Trees's performance can be similar or even slower than HyPer, because the advantage of column-store is more significant than index nesting. However, P-Trees still achieve better parallel performance in these queries, indicating that our implementation potentially allows for better scalability.

Using 144 threads, P-Trees achieve a 69× speedup on average, while HyPer and MemSQL get speedup around 30×. This indicates that the good parallel performance of P-Trees benefits greatly from better scalability.

Parallel Performance Gain Breakdown. To understand the performance gain of our TPC-H implementation, we look at three optimizations we use: the index nesting, the secondary indexes, and whether we include the table entries *inline* in the index (as with index organized tables [256]), or in a separate record elsewhere in memory with a pointer to it. We implement 12 representative queries in TPC-H, that use different tables and indexes in our implementation. We start from plain P-Trees with no optimization, and add the three optimizations one by one to test the improvement. Results are shown in Table 11.3. We mark the test cases that P-Trees are better than *both* HyPer and MemSQL as grey cells.

For data inline, the performance gain is about 12%. This means that storing pointers in indexes to save memory overhead caused by index nesting causes about 12% overhead in queries. Even using plain P-Trees, our implementation outperforms both HyPer and MemSQL in four out of 12 tested queries.

	P-Tree	P-Tree		P-Tree		P-Tree		Hyper	MemSQL
Nested	no	no		no		**yes**		-	-
Sec. Idx	no	no		**yes**		yes		**yes**	-
Inline	no	**yes**		yes		yes		-	-
	time	**time**	**ratio**	**time**	**ratio**	**time**	**ratio**	**time**	**time**
Q1	388	317	22.3%	324	-2.1%	324	0.0%	334	375
Q2	59	52	12.9%	52	0.0%	18	190.4%	251	233
Q3	1053	985	6.9%	985	0.0%	192	414.1%	464	2377
Q4	630	517	22.0%	417	24.0%	36	1072.6%	363	403
Q5	613	531	15.5%	444	19.4%	69	544.2%	520	1171
Q6	395	325	21.8%	53	516.8%	53	0.0%	99	230
Q8	1044	984	6.2%	984	0.0%	116	749.5%	125	298
Q11	49	52	-6.1%	52	0.0%	14	269.3%	67	120
Q12	505	472	6.9%	104	355.6%	104	0.0%	120	277
Q13	794	777	2.1%	777	0.0%	403	92.7%	1595	4561
Q14	409	337	21.4%	23	1350.0%	23	0.0%	82	250
Q18	1998	1718	16.3%	1718	0.0%	421	307.9%	3174	6327
Gmean	446	398	12.0%	234	70.2%	82	184.2%	286	585

Table 11.3: Breakdown of the P-Tree's performance gain on TPC-H – The running time of using P-Tree with different optimizations enabled, including index nesting (Nested), secondary index (Sec. Idx) and data inline (Inline). "Ratio" means the improvement compared with the previous column. Experiments are on SF 100 with 144 threads. We highlight (the grey cells) all P-Trees' throughput numbers that outperform *both* HyPer and MemSQL.

Five out of 12 tested queries achieve significant improvement from secondary indexes. The improvement ranges from 24% to 1400%. Even without index nesting, The P-Tree is better than both HyPer and MemSQL in nine out of 12 queries.

Eight out of 12 tested queries take advantage of tree nesting. On average the improvement of using index nesting is up to 10×, with an average of 2×.

In summary, the major gain of P-Trees' good query performance comes from index nesting. Some queries such as Q1 require to scan all lineitems, which cannot benefit from index nesting. In this case, P-Trees with all optimizations can only achieve similar performance to a plain P-Tree, as well as HyPer and MemSQL. For the others, the overall improvement of all optimizations can be up to 20× faster than the plain P-Trees.

11.6.3 HTAP Workload Evaluation

We test P-Trees on the hybrid benchmark TPC-HC (see Section 11.6.1). We run our system with durability enabled, and stop queries and transactions after 10^5 transactions.

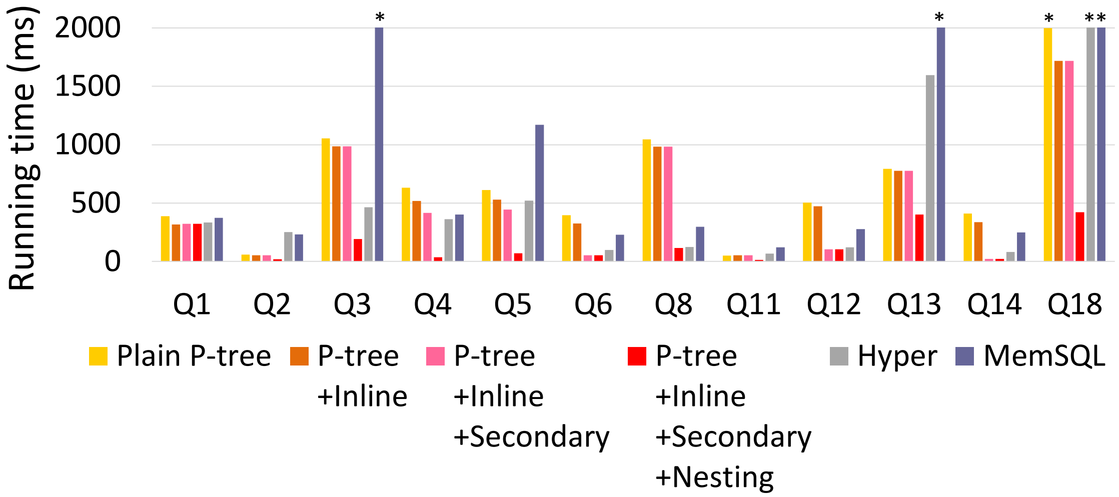


Figure 11.6: Breakdown of the P-Tree's performance gain on TPC-H – 144 threads, SF 100. "*" means the bar exceeds the y-axis.

We show the running time on queries in Table 11.2 in the columns labeled with ✓, and report the update throughput in Table 11.4.

We compare our results to HyPer and MemSQL. The throughput numbers of HyPer are not satisfactory, which are 100× slower than MemSQL and our P-Tree, and are much slower than they report in their paper [178] (we note that the benchmark is slightly different, and our workload is 100× larger than that in [178]). In our evaluation, we test both the Tableau version of HyPer as well as its academic predecessor. The old version did not support JDBC, so we only report the numbers from the Tableau version. However, we found that both versions had comparable performance on TPC-H SF 100. Through correspondence with the original HyPer authors, we understand the Tableau version does not have certain features enabled. We suspect that this contributes to the degradation of performance observed.

Update Overhead. As shown in Table 11.2, for P-Trees, adding a sequential update process only causes about 5% overhead to the TPC-H queries, which is lower than both HyPer and MemSQL. The slowdown is likely caused by the contention in updating the reference counters, which will invalidate cache lines. We note that there are a small number of queries that have higher throughput when running with updates. This is possible that when running with updates, the query coincidentally gets better cache locality.

P-Tree Update Throughput. Our DBMS achieves a throughput of 12k update transactions per second on a single thread. For the Payment transactions on P-Trees, the only updated tree is the index for customers. Thus the throughput is the highest among the three. Each New-Order transaction updates five trees. Therefore it is much slower (about 8×) than the Payment transactions. Finally the Delivery transaction is the most expensive among the three, not only because it needs to update an average of five orders, but it also updates six trees in total. Therefore, it is about 5× slower than the New-Order transactions.

Update Throughput Comparison with MemSQL. Overall P-Trees have almost exactly the same performance as MemSQL in updates. On New-Order it is about 3× slower, while on Payment it is about 3× faster, and about 2× faster on Delivery. The extra cost on New-Order is due to the extra tables that have to be updated.

11.6.4 Memory Overhead Analysis

In this section, we analyze the memory overhead caused by path-copying and index nesting in our TPC-H and TPC-HC experiments.

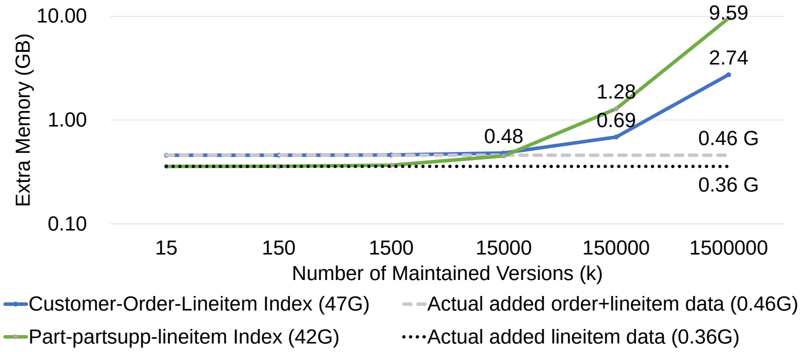


Figure 11.7: TPC-HC Memory Overhead – The memory overhead of executing 1.5M New-Order transactions for the TPC-H benchmark (SF=100), which is about 1% of the original data. The x-axis shows the numbers of kept versions k. We show numbers about two indexes in our system: the customer-order-lineitem index, and the part-partsupp-lineitem index. The original size of customer-order-lineitem is about 47G, and the part-partsupp-lineitem index is about 42G.

Index Nesting Memory Overhead. Index nesting can cause extra space when a table is involved in multiple logical hierarchies or secondary indexes. One can expect space-performance tradeoff by designing different nestings of indexes. We now analyze the memory usage in our TPC-H nested index implementation.

In total, to store 100G TPC-H raw data, our implementation uses 265G data, including the raw data, metadata for indexes (e.g., pointers and subtree sizes for each node), and duplicates caused by index nesting. This number matches the theoretical estimation of memory usage in Table 11.5 in the second last column *Inline*. This gives us the good performance as we report in Table 11.2. One can save space by storing only pointers in indexes, avoiding physically copying data. For our TPC-H implementation, this saves about 55G data, as we show in the last column *Indirect* in Table 11.5. However using indirections may cause extra cache misses for queries. This overhead in query time is about 12% as we show in Table 11.3. We can also save space by storing fewer indexes. For example, if we drop the secondary index on `receiptdate`, we can save about 30G memory. The only side effect is the slowing down of Q12 by 4×.

Both HyPer and MemSQL use about 100G memory on 100G TPC-H raw data. We do not use any compression as they do. We note that there are also ways to compress P-Trees [122].

Path-copying Memory Overhead. We next measure the memory overhead for maintaining multiple snapshots using P-Trees through path-copying. We present our results in Figure 11.7. We execute 1.5×10^6 New-Order transactions (1% of the original database size). For simplicity, we only evaluate two indexes in our system: the customer-order-lineitem nested index T_{cust}, and the part-partsupp-lineitem nested index T_{part}. We keep k (ranging from 15 to 1.5M) recent versions, and garbage collect other versions. We present the peak memory footprint over time in Figure 11.7. The grey dotted line shows the actual memory size required by the new orders. We do not count the memory to store strings because they are not stored in the indexes, and we never copy them. The extra used memory includes the added data itself, metadata to maintain the trees (e.g., pointers in each node) as well as extra space caused by path-copying.

To add a new order O_n from customer C_n in T_{cust}, we first build a tree of all lineitems in O_n, attach it to a new created order node, and insert this node into the inner tree of C_n. We use nested copying in Section 11.4. Each lineitem creates exactly one lineitem node because they are not inserted into existing trees, but a newly-built tree. Therefore there is no extra copying of lineitem nodes. Orders are stored in small inner trees of the customers. Therefore the overhead in copying order nodes is also small because each new order only creates about 3-4 order nodes. The major overhead occurs in copying customer nodes. Every new order copies a path of customers. However, when k is small, the out-of-date customer nodes all get collected due to precise garbage collection, leaving only small memory overhead (less than 0.02G). In fact, the used space is almost exactly $k \cdot h_c \cdot s_c$, where $h_c \approx 23$ is customer tree height, and $s_c = 64B$ is the customer node size. However, if more versions are kept, the memory overhead is more than 4×.

In T_{part}, every new lineitem inserted leads to path-copying on all three levels of nesting, causing more overhead than T_{cust}. The order data are not added to this index. The total memory overhead is generally low when old versions are collected in time. Similarly, the lineitem and partsupp trees are all shallow inner trees. Thus the overhead mainly comes from a large number of copied part nodes especially when k is large.

In both cases, index nesting helps to reduce memory overhead from path-copying because the dominate memory usage, which is the lineitems, are kept only in small shallow trees. In fact, index nesting shifts the copying of inner tree nodes to outer tree nodes. In our indexes, many of the outer tree nodes have to be updated and copied anyway because of the New-Order transactions (e.g., the available quantity of a partsupp). Therefore the total number of copied nodes reduces because of index nesting.

	New-Order	Payment	Delivery	Overall
MemSQL	23,655	25,156	765	10,280
Hyper*	67	214	11	75
P-Tree	7,037	61,332	1,110	8,696

Table 11.4: Update Throughput on TPC-HC – Updates are executed sequentially, while queries are executed one by one using multi-cores. *: HyPer's performance is below expectation. More explanation is given in Section 11.6.3.

	Bytes per entry			Dup-	Size	Memory (GB)	
	String	Metadata	Data	licates	(M)	Inline	Indirect
Lineitem	79	24	40	4	600	187.20	138.02
Order	109	40	32	3	150	45.40	39.81
Customer	207	40	24	1	15	3.79	3.90
Partsupp	199	40	24	2	80	24.36	23.77
Part	148	40	24	1	20	3.95	4.10
Supplier	181	40	24	1	1	0.23	0.24
Total						**265G**	**210G**

Table 11.5: Memory usage of our TPC-H implementation – SF=100. This estimation matches our experimental measurements. "String" is the size of string per entry, "Data" means the size of other fields per entry. Strings are not copied in duplicates.

In summary, the space overhead of P-Tree is small when GC is in time or when the change of the DBMS is not significant, but will become a big bottleneck if more versions remain in memory. Usually it is unlikely to have the number of living version more than the number of physical threads. In our case, the overhead of keeping 144 versions is rather low. If we want to keep all history, we can also write old versions to disk to collect them in memory.

Chapter 12

Version Maintenance for Concurrent Systems

12.1 Introduction

In a concurrent system, concurrent read-only transactions access the latest state of the system, and concurrent write transactions update the latest state of the system and commit. Such a system is useful in database management systems (DBMS) [61, 181, 214, 226] software transactional memories (STM) [229, 237], operating systems [206, 207], etc.

In general, it is usually hard to achieve solutions with tight bounds and good properties on arbitrary transactions. This can be both due to the overhead of the transactional system, and due to inherent dependences among the transactions, forcing the system to wait for another to complete. When most transactions are read-only, however, the prognosis is significantly better. In particular, read-only transactions (readers) can in principle proceed with constant delay and without delaying any writing transactions (writers), since they do not modify any memory, and hence other transactions do not depend on them. This can be very useful in workloads dominated by readers. Several approaches try to take advantage of this. Read-copy-update (RCU) [206] allows for an arbitrary number of readers to proceed with constant delay, and has become a core idiom widely used in Linux and other operating systems [207]. In RCU, however, readers can arbitrarily delay (block) a writer, since a writer cannot proceed until all readers have exited their transaction. This is particularly problematic if some readers take significant time, fault, or sleep [201]. Indeed RCU in Linux is used in a context in which the readers are short and cannot be interrupted.

With multi-versioning [61, 181, 214, 226, 229, 237], on the other hand, not only can readers proceed with constant delay, but in principle, they can avoid delaying any writers—a writer can update a new version while readers continue working on old versions.

Therefore a single writer and any number of readers should all be able to proceed without delay (multiple writers can still delay each other).

Multi-versioning, however, has some significant implementation issues that can make the “in principle” difficult to achieve in “theory” or “practice”. One is that memory can become an issue due to maintaining old versions, possibly leading to unbounded memory usage. Ideally one would like to reclaim the memory used by a version as soon as the last transaction using it finishes. The detection of such out-of-date versions is defined as the *version maintenance* (VM) problem. Ben-David et al. [50] propose a framework to VM problems. Detecting when a version is safe to collect, this framework requires an underlying functional data structure which supports appropriated garbage collection. In this chapter, we are interested in studying how P-Trees can be used in this framework. We will also present a simple lock-free algorithm for the VM problem that can support effective multi-version concurrent system.

The property that P-Trees support snapshotting using path-copying and a correct GC makes P-Trees good candidates for a VM solution. Combining P-Trees with the lock-free VM solution yields a multi-versioning transactional system that is lock-free, serializable, guaranteeing no-abort for all readers and one writer, and with safe and precise GC.

At the end of this chapter, we will discuss how P-Trees can be combined with other VM solutions in previous work, which is studied in [50]. These solutions include hazard pointers [210], epoch-based GC [111], Read-Copy-Update (RCU) [206, 207], and a wait-free safe and precise (WFPS) VM solution [50].

12.2 Preliminaries

We consider *asynchronous shared memory* with P processes. Each process p follows a deterministic sequential protocol composed of *primitive operations* (read, write, or compare-and-swap) to implement an object. We define *objects, operations* and *histories* in the standard way [157]. We consider *linearizability* as our correctness criterion [156, 158]. An *adversarial scheduler* determines the order of the invocations and responses in a history. We refer to some point in a history as a *configuration.* We define the *time complexity* of an operation to be the number of instructions (both local and shared) that it performs. Note that this is different from the standard notion of *step complexity* which only counts access to shared variables.

Transactions. We consider two types of transactions: *read-only* and *write*. Each transaction has an *invocation*, a *response*, and a *completion*, in that order. A transaction is considered *active* between its invocation and response, and *live* between its invocation and completion. Intuitively, the transaction is executed between its invocation and response, and does some extra ‘clean-up’ between its response and its completion. We require that transactions be strictly serializable, meaning that each transaction appears to take effect at some point during its active interval. We refer to a write transaction as *single-writer* if no other write transaction is live while it is live.

Functional Data Structures. Although in this thesis we only use P-Trees as the underlying VM data structure, the framework used in this chapter is applicable to general functional data structures, which can be an arbitrary DAG instead of just trees. We assume that the memory shared by transactions is based on purely functional (mutation-free) data structures. We use the PLM as defined in Section 2.3. Using PLM instructions, one can create a DAG in memory, which we refer to as the *memory graph*.

We define the *version root* as a pointer to a tuple, such that the data reachable from this tuple constitutes the state that is visible to a transaction. Then each update on version v yields a new version by path-copying starting from the version root of v, and the new copied root provides the view to the new version. In our framework, every transaction t acquires exactly one version $V(t)$. If t has not yet determined its version at configuration C, then $V_C(t) = null$ until it does. We use the version roots as the data pointers in the Version Maintenance problem.

Garbage Collection. We assume all tuples are allocated at their tuple instruction, and freed by a `free` instruction in the GC. The *allocated space* consists of all tuples that are allocated and not yet freed. For a set of transactions T, let $R(T)$, or the *reachable space* for T in configuration C, be the set of tuples that are reachable in the memory graph from their corresponding version roots, plus the current version c, i.e. the tuples reachable from $\{V(t)|t \in T\} \cup \{c\}$.We say that a tuple u *belongs* to a version v if u is reachable from v's version root. Note that u can belong to multiple versions. We define a precise and a safe GC, respectively, as follows.

Definition 12. *A garbage collection is* precise *if the allocated space at any point in the user history is a subset of the reachable space $R(T)$ from the set of live transactions T.*

Definition 13. *A garbage collection is* safe *if the allocated space is always a superset of the reachable space from the active transactions.*

Roughly speaking, precise GC means to free any out-of-date tuples in time, and safe GC means not to free any tuples that are currently used by a transaction.

12.3 The Version Maintenance Problem

The VM problem is defined in [50], which is to implement a linearizable object with three operations: *acquire*, *release* and `set`. The *acquire* operation returns a handle to the most recent version, in a way that ensures it cannot be collected. The `set` operation updates the current version to a new pointer, returning whether it succeeded or failed. The *release* operation indicates that the currently acquired version is no longer needed by the process, potentially making it available to be collected. It returns a list of versions that can be collected—i.e., for which no other process has acquired it and not released it. Only

Read Transaction

```
1 v = acquire(k);
2 user_code(v);
3 // response
4 versions = release(k);
5 for (v in versions) collect(v);
```

Write Transaction

```
1 v = acquire(k);
2 newv = user_code(v);
3 flag = set(newv);
4 // response if successful——— update visible here
5 versions = release(k);
6 for (v in versions) collect(v);
7 if (!flag) collect(newv) and retry or abort
```

Figure 12.1: Read and Write transactions with *acquire*, `set`, and *release* – k is the process ID.

one version can be acquired on any process at any time, i.e. the current version must be released before a new one is acquired. In the *precise* VM problem, the release will return a singleton list precisely when the process is the last to release its version, and an empty list otherwise. We give a solution to the precise version.

The VM object can be used to implement read-only and writing transactions as shown in Figure 12.1. The read transaction is effectively done after step 2 (response could be sent to a client), and the rest is a cleanup phase for the purpose of GC. Similarly, writing transactions are done after step 3, at which point the result is visible to other transactions. After the release, any garbage can be traced from the released pointers and collected in work linear in the amount of garbage collected using a standard reference counting collector.

We refer to the pointer to a version as the *data pointer*. More formally, if d is a pointer to data, `set`(d), if successful, creates a new version with pointer d and sets it as the *current version*, i.e.,

Definition 14. *The* current version *is defined as the version set by the most recent successful* `set` *operation.*

The operations are intended to be used in a specific order: an *acquire*(k) should be followed by a *release*(k), with at most one `set`(k, d) in between, where d is a pointer to a new version. If this order is not followed for each k, then the operations may behave arbitrarily; that is, we do not specify a 'correct' behavior for the operations of a Version

Maintenance object O in an execution once any operations are called out of this order on O.

We define the liveness of a version v as follows.

Definition 15. *A version v is* live *at time t if it is the current version at t, or if $\exists k$, s.t. an* acquire*(k) operation A has returned v but no* release*(k) has completed after A and before t.*

We note that a version is live while a transaction using that version is active. The transaction itself can remain live after its version is dead, while it garbage collects.

The following is the sequential specification of these operations assuming that they are called in the correct order (*acquire-release* or *acquire*-`set`-*release* for each id k).

- `data* acquire(int k)`: Returns the current version.
- `data** release(int k)`: Returns a (possibly empty) list of versions that are no longer live. No version can be returned by two separate *release* operations.
- `bool set(int k, data* d)`: Sets the version pointed to by d as the current version. Returns *true* if successful. May also return *false* if there has been a successful `set` between this `set` and the most recent `acquire(k)`. If the `set` returns *false*, it has no effect on the state of the object.

In the proofs of linearizability of our algorithm (which we show in Section 12.4), we state linearization points, and then proceed to show that for any given history, if we sequentialize it based on the stated linearization points, it adheres to the above sequential specification.

We say that a process p_k has *acquired* version v if *acquire*(k) returns v, and say p_k has *released* v when the next *release*(k) operation returns. If a `set` operation returns *true*, we say that it was *successful.* Otherwise, we say that the `set` was *unsuccessful* or that the `set` *aborted.* Note that conditions for correct aborting for the `set` are reminiscent of 1-abortability defined by Ben-David *et al.* [47], but we relax the requirements to allow a successful `set` to cause other `sets` to abort even if it was not directly concurrent with them, but happened sometime since that process's last *acquire*.

An implementation of a Version Maintenance object is considered *correct* if it is linearizable as long as no two operations with the same input k run concurrently. The GC is *safe* to invoke on the versions *release* returns if for any particular version v, there is exactly one *release* operation that returns true. Furthermore, it is considered *precise* if the *release* operation returns exactly the versions that stop being live at the moment the *release* operation returns. Note that this means that in a precise implementation of the Version Maintenance problem, each *release* operation r returns a list containing at most one version. This version must be the one that r released, and r must be the last operation done on v.

It is useful to note the following two facts (proved by [48, 50]), which must hold in any algorithm that solves the version maintenance problem.

Observation 12.3.1. *v is live immediately before the linearization point of a* release_v *operation.*

Observation 12.3.2. *A version v is live for a contiguous set of configurations.*

Where convenient, for a version v, we use $\textit{acquire}_v$, $\textit{release}_v$ and $\texttt{set}_v$ to denote an *acquire* operation that acquires v, a *release* operation that releases v, and a `set` operation that sets v as the current version, respectively.

12.3.1 A Lock-free Algorithm

In this section, we describe a simple lock-free algorithm for the VM problem. We will later show that just combining this simple algorithm with P-Trees gives an effective solution to a multi-versioned concurrent system. The algorithm uses a hash table S to maintain the status of all live versions. Similarly, a hash table D is used to store the data pointers associated with the live versions. Each non-empty element of S stores a version along with the count of that version. Intuitively, the count of each version represents the number of processes working on that version. We represent a version v as a timestamp plus an index. The index tells us which element of D and S is reserved for that version. A is an announcement array mapping each processor id to the version it is working on. We use hash tables to maintain information on all live versions. Note that at any time, there can be at most P live versions, so both hash tables have size $2P$. We use a global variable V to represent the current version.

Acquire. The acquire operation reads the most recent version from V, increments the count of that version, and returns the data pointer associated with it. Incrementing the counter is implemented by performing a CAS in a loop. Every time the CAS fails, the operation re-reads the current version and the status of that version. Before returning, an *acquire*(k) operation writes the version that it acquired into its persistent local variable $A[k]$. This is so that the next call to *release*(k) knows which version to release. Note that the version number i that a processor p acquires is not necessarily the current version (except for the acquire of the writer, which will always return the current version) when it returns, but is still a valid version whose counter value is greater than 0, so that it will not be collected by others before p terminates.

Set. The `set`(k, d) function first randomly finds an empty slot i for the new version in the hash table S (the hash table is accessed using linear probing). Then it creates a new version v using index i and the current timestamp plus one. Then it stores v into $S[i]$ with count 0. Because there can be concurrent writers taking the slot i, a CAS is used to declare i for this writer. If the CAS fails (which means that other writers takes this slot during the above two lines), the `set` operation fails. Otherwise, it writes d into $D[i]$ and sets v as the current version. Finally, it checks if any other writers have changed V after it *acquire*s the version. If not, the new version can be committed, otherwise, the `set` fails. The order

```
struct Version{ int timestamp;  int index; };
struct VersionStatus{ Version v; int count; };
Version V;
VersionStatus S[2*P];
Data* D[2*P];
Version empty = ⟨-1, -1⟩;
Version A[P]; //Persistent local vars.

Data* acquire(int k) {
  Version v, VersionStatus s;
  do {
    v = V;
    s = S[v.index];
  } while (!CAS(S[v.index], ⟨v,s.count⟩, ⟨v,s.count+1⟩));
  A[k] = v;
  return D[v.index]; }

bool release(int k) {
  Version v = A[k]; bool last = false;
  do {
    VersionStatus s = S[v.index];
    if(s.count == 1) last = true;
  } while (!CAS(S[v.index],s,⟨s.v,s.count-1⟩));
  if (v == V || !last) return false;
  VersionStatus s = S[v.index];
  if(s.v != ⟨v, 0⟩) return false;
  return CAS(S[v.index],s,⟨empty,0⟩); }

void set(int k, Data* d) {
  int index = random_hash(0, 2*p);
  while (S[index] != ⟨empty, 0⟩) index = (index+1)%(2*p);
  Version v = ⟨V.timestamp+1, index⟩;
  bool f = CAS(S[index], ⟨empty,0⟩, ⟨v, 0⟩);
  if (!f) return false;
  D[index] = d;
  bool result = CAS(V, A[k], v);
  if (!result) S[index] = ⟨empty,0⟩;
  return result;}
```

Figure 12.2: A lock-free algorithm for the Version Maintenance Problem.

matters since whenever the `cur_idx` is acquired by other threads, the whole version must be ready, and whenever the flag is read as *true*, the data structure should be ready.

Release. Basically, a *release*(k) operation decrements the counter of the corresponding version v, and collect the corresponding version if necessary. Intuitively the *release* function returns true iff. it does the garbage collection of its corresponding version (will be proved later). The *release*(k) operation first reads $A[k]$ to check which version it's responsible for decrementing. In our algorithm, the last one that releases this version is always responsible to collect it. Then it tries to decrement the counter with a CAS, just like an *acquire* operation, retrying until successful. After the decrement, if v is still the current version or if the count of v is non-zero then the operation returns *false* because v is still live. Otherwise, the count of v is 0 and v is no longer the current version, so the *release*(k) operation tries to make v unacquireable by setting $S[v.index]$ to empty with a CAS. If it succeeds, then v is no longer live and it returns *true*. Otherwise, either some other operation has set $S[v.index]$ to empty or some *acquire* operation has incremented the count of v. In the first case, the *release*(k) operation returns *false* because some other *release*$_v$ operation has already returned *true*. It also returns *false* in the second case because v is still live.

12.4 The Correctness of the Lock-free Algorithm

In this section, we show that our lock-free algorithm is linearizable. The result can be summarized as Theorem 12.4.1. We defer the full proof to the supplement material.

Theorem 12.4.1. *Algorithm 12.2 is a linearizable solution to the Version Maintenance problem.*

We first present the linearization points for each operation in our lock-free algorithm.

Definition 16. *For each operation op, its* linearization point *is as follows:*

- *If op is a successful* `set`*, then it is linearized at line 33. This is the line that updates* V*. If op is an unsuccessful* `set`*, it linearizes at its return.*
- *If op is an* acquire*, then it is linearized on line 12 of the loop that performs a successful CAS on line 14.*
- *If op is a* release*, then there are two cases: (1) If op completes and returns* true*, then it is linearized at its final step. (2) Otherwise, it is linearized when it performs a successful CAS on line 24.*

We then prove that these linearization points satisfy the sequential specification outlined in Section 12.3, which them proves Theorem 12.4.1.

We refer to $S[v.index]$ as the counter of v. Intuitively, *release* operations returning *false* are linearized when they decrement the counter of the version they are trying to release.

Also, we linearize *acquire* operations at when it reads the version it later successfully grabs. We first prove the following two helper lemmas for better understanding the algorithm.

Lemma 12.4.2. *When the counter of a version v is more than 0, this version is live.*

Proof. The counter of a version v is more than 0 only if line 14 of an $\mathit{acquire}_v$ operation increments the counter and the counter has not yet been decremented by line 23 of the corresponding $\mathit{release}_v$ operation. Since $\mathit{acquire}_v$ operations are linearized before they increment the counter and $\mathit{release}_v$ operations are linearized either when they decrement the counter or at some later step, we know that v is live whenever its counter is more than 0. □

Lemma 12.4.3. *For each version v, there is at most one* $\mathrm{release}_v$ *operation that returns* true.

Proof. A $\mathit{release}_v$ operation returns *true* only if the CAS on line 27 changes $S[v.\mathit{index}]$ from $\langle v, 0\rangle$ to $\langle \mathit{empty}, 0\rangle$. The lemma holds because $S[v.\mathit{index}]$ will never change back to $\langle v, 0\rangle$ after it has been set to $\langle \mathit{empty}, 0\rangle$. □

Now we are ready to show that the linearization points satisfy the sequential specifications. Lemma 12.4.4 proves the first part of the sequential specifications, which says that *acquire*() operations return the correct value. Together, Lemmas 12.4.5 and 12.4.6 prove the second part of the sequential specification, which says that a $\mathit{release}_v()$ operation returns 1 if and only if v changes from being live to not live.

Lemma 12.4.4. *An* acquire*(k) operation Q returns the pointer written by the last* `set`*() operation linearized before it.*

Proof. Note that at any point in an execution, the global variable V stores the version created by the last `set`() operation linearized before this point. Furthermore, at any point in the execution, $D[V.\mathit{index}]$ stores the pointer written by the last `set`() operation linearized before this point. This is because $D[i]$ cannot change as long as $S[v.\mathit{index}] \neq \langle \mathit{empty}, 0\rangle$ and $S[v.\mathit{index}]$ cannot be set to $\langle \mathit{empty}, 0\rangle$ as long as $i = V.\mathit{index}$ (due to the check on line 24).

Let v_0 be the value of V at Q's linearization point. It's easy to see that Q increments the count of v_0 by looking at how *acquire* operations are linearized. While the count of v_0 is non-zero, $S[v_0.\mathit{index}]$ cannot be set to $\langle \mathit{empty}, 0\rangle$ because the CAS on line 27 of $\mathit{release}_{v_0}$ only succeeds when the count of v_0 is zero. Therefore $D[v_0.\mathit{index}]$ cannot be changed as long as the count of v_0 is non-zero, so Q returns the data pointer associated with v_0. □

Lemma 12.4.5. *If a* $\mathrm{release}_v$*() operation R returns* true*, then v is live before the linearization point of R and not live after.*

Proof. By Observation 12.3.1, v must be live before the linearization point of R.

We first show that any *acquire*$_v$ operation linearized before R must have increased the counter of v on Line 14 before the linearization point of R. Note that after the linearization point of R, $S[v.index].v = empty$, meaning that the counter of v can only increment before the linearization point of R. Therefore all *acquire*$_v$ operations must increment the counter of v before the linearization point of R.

Next, we show that all *acquire*$_v$ operations linearized before R have corresponding *release*$_v$ operations that are also linearized before R. We know that $S[v.index] = \langle v, 0\rangle$ immediately before the linearization point of R, so each *acquire*$_v$ operations linearized before R has a corresponding *release*$_v$ operation that decrements the count for v before the linearization point of R. By Lemma 12.4.3, all *release*$_v$ operations other than R return *false* and are linearized when they decrement the count for v. Therefore all *acquire*$_v$ operations linearized before R have corresponding *release*$_v$ operations that are also linearized before R.

We can see from the code that v is not the current version at the linearization point of R because R passed the check in line 24. Combining this fact with the previous fact that we showed, we have that v is not live after R. □

Lemma 12.4.6. *If a* release$_v$*() operation R returns* false*, then v is live after the linearization point of R.*

Proof. Recall that R is linearized at CAS on line 23 that decrements the count of v. By Observation 12.3.1, we know that v is live just before this CAS. Therefore to show that v is live immediately after this CAS, it suffices to show that v is live at some configuration after this CAS (By Observation 12.3.2).

If R returns on line 24, then either $v = V$, in which case v is live by definition, or the count of v is non-zero immediately after the linearization point of R, in which case v is live by Lemma 12.4.2.

If R returns on line 26 or line 27, then either some *release*$_v$() operation R' is linearized before the end of R or the count of v is non-zero at some point after the linearization point of R. In the later case, we know that v is live after the linearization point of R by Lemma 12.4.2. In the former case, we know that R' must be linearized after R because v is live before the linearization point of R and v is not live after the linearization point of R' (by Lemma 12.4.5). By Observation 12.3.1, v is live immediately before the linearization point of R', so v must have been live after the linearization point of R. □

Together, Lemmas 12.4.4, 12.4.5 and 12.4.6, and Definition 16 directly imply Theorem 12.4.1.

However note that this algorithm is not wait-free, and does not have time bounds. In particular, the while loop in *acquire* can run arbitrary long. This is because of the infinite loop in updating counters.

12.5 Wrapup

Ben-David et al. have shown that given a linearizable VM object, e.g., the lock-free algorithm in Section 12.3.1, one can achieve a serializable solution to the VM problem. As a result, plugging in the lock-free algorithm gives a simple and correct solution of the VM problem.

The correctness of the VM problem means that on the version level, a old version can be identified as garbage once the last user releases it. To achieve tuple level preciseness, this requires extra properties of the underlying VM data structure. Actually, as long as the functional data structure supports a *correct* (defined in Section 5.2) *collect* algorithm, one can achieve *safe* and *precise* garbage collection for the corresponding VM solution [50].

Intuitively, a linearizable precise VM solution provides an interface for safe and precise garbage collection over *versions*, since $release_v$ returns true if and only if it is the last usage of v. However, the precision and safety on the granularity of *tuples* relies on a “correct” *collect* operation, which, intuitively, should free all nodes that are no longer reachable as soon as possible. In conclusion, combining the lock-free algorithm with P-Trees can achieve a concurrent multi-version system that is serializable, lock-free, no-abort for all readers and one writer, and safe and precise GC.

In [50], the authors show that P-Trees, when combining with other VM solutions, can achieve even better properties. In particular, based on the results and experiments in [50], we can conclude:

- Combining P-Trees with a hazard-pointer-based VM object, we can obtain a solution that maintains $2P$ versions, and achieves high throughput in queries. The GC will not be precise.
- Combining P-Trees with an RCU-based VM object, we can obtain a solution that maintains exactly one version, but with lower throughput for writers. The GC will be precise, but the writers can be blocked.
- Combining P-Trees with the precise, safe and wait-free (PSWF) algorithm, we can obtain a solution that maintains at most P versions (on average can be much fewer), and achieves good performance considering both readers and writers. All readers can be delay-free (defined in [50]), and at least one writer only suffers from $O(P)$ delay. All transactions are lock-free. The GC will be precise.

Chapter 13

Inverted Index Searching

13.1 Ranked Queries on Inverted Indices

13.1.1 General Design

Our last application of augmented maps is building and searching a weighted inverted index of the kind used by search engines [236, 276] (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 k results with highest weight, as a search engine would list on its first page.

This can be represented rather directly in our interface. The inner map, maps document-ids (D) to weights (W) and uses maximum as the augmenting function f. The outer map maps terms (T) to inner maps, and has no augmentation. This corresponds to the maps:

$$M_I \;= \mathbb{AM}\,(\,D, <_D, W,\;\; W, (k, v) \rightarrow v, \max_W, 0\,)$$
$$M_O = \mathbb{OM}\,(\,T,\; <_T \;\; M_I, \,)$$

In the implementation, we use the feature of PAM that allows passing a combining function with *union* and *intersection* (see Chapter 3), for combining weights. Note that an important feature is that the *union* function can take time that is much less that the size of the output. 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.

With such an inverted index, queries for a conjunction of words (**and**) can be implemented by looking up the posting list for each word, and then taking an intersection of the corresponding sets. Similarly, a disjunction (**or**) is the union of the sets, and difference can be used to exclude documents with a given term (**and-not**). Because of the importance of inverted indices in document and web search there has been significant research on developing efficient implementations of set operations for this purpose, e.g., [38, 165, 241, 243]. As discussed in previous papers, it can be important for unions and intersections to take time that is sublinear in the sum of the sizes since the size of the lists can be very different—e.g. consider the lists for "set" (5.8 billion hits in Google) and "cacomistle" (54 thousand hits in Google). PAM is well suited for such inbalanced queries since the time is mostly a function of the smaller size.

It is most often useful not just to report a flat set of documents that satisfy the query, but instead to report a reasonably small subset of the most important documents. Here we discuss how augmented maps can be used to efficiently implement such queries. The idea is to keep a *document-word weight* with each document-word pair. This weight represents a combination of the importance of the word in the document and the importance of the document in the corpus. The importance of the word in a document is often measured by the *term frequency-inverse document frequency* (tf-idf) [236], which measures how frequently a word appears in a document relative to how frequently it appears in the corpus as a whole. In practice, it is also important to consider where the word appears in a document, giving words in titles, associated with an anchor, or near the beginning a heavier weight. The importance of the document can be measured by algorithms such as PageRank [224]. The importance of the word in the document and document in corpus can then be combined to give an overall document-word weight. A zero weight is given to document-word pairs that do not appear in the corpus so the representation can be kept sparse.

The weights can be maintained in an inverted index by augmenting each posting list so it is not just a set of documents, but a map from documents to weights, and further augmenting the maps to keep the maximum weight in the map. When taking unions or intersections the weights can be added for documents that appear in both sets. The map returned by a query is therefore a mapping from documents that satisfy the query to weights corresponding to importance of that document relative to the query. In this way, returning the document with maximum weight will return the most important document relative to the query in constant time. The next k most important documents can be found in $k \log n$ time (n is the size of the posting list). For example, a query could be of the form ("posting" **and** "file") **or** ("inverted" **and** "index") with the objective of returning the most relevant documents. The number of documents that match the query could be large given that the words have multiple meanings, but the number relevant documents might be small.

Figure 13.1 gives the code for implementing such an augmented map in PAM. Words are stored as character strings (`char *`), document identifiers are integers, and weights are floats. A posting list is an augmented map using the maximum of the weights (defined in a_map). The build_reduce function builds a map from a sequence of key-value pairs (not necessarily sorted) and uses the supplied function to combine the values of equal keys. In our case the documents with equal words (keys) should be combined by creating a posting list (post_list) over them, and combining weights of equal words within a document with add).

The implementation uses at least four important features of PAM. Firstly, when taking unions and intersections it uses the combining function to specify how the weights of the common documents that appear in both document sets should be combined. Secondly, augmenting the map with the maximum weight document in the result of the query allows it to quickly find the most important documents. Thirdly, PAM takes time that is roughly proportional to the smaller list, making it efficient when the posting lists have significantly different sizes. And, fourthly, it makes use of persistence since the input posting lists (maps) are not destroyed, but instead a new list is created when applying unions or intersections. Note that supporting persistence by copying both input lists would be very inefficient if one is much larger than the other (PAM will only take time mostly proportional to the smaller).

13.1.2 Concurrent Queries and Updates

Because of the features of P-Trees, we can also support dynamic updates in such inverted indexes, i.e., new documents are added to the corpus, and some of the old ones are removed. Simultaneously multiple users are querying on the index. Usually updates are conducted by the server, and can be easily wrapped in one write transaction. In addition, adding one document means a large set of term-document relations added to the database, and we want a whole document is combined into the database *atomically*, i.e., the queries will never read a partially updated document in the database. To guarantee serilizability, we only use a single global writer. The correctness would be supported by the functional tree structure. We note that even though we only use one global writer, it involves adding a set of term-doc pairs, which can also run in parallel.

Updating an inverted index can be supported effectively by the PAM interface for *build*, and *union* allowing for complementary functions. Assume the original index is m. To add a set of documents, we first convert it to a list of key-value pairs $l = \{(t_i, (d_i, w_i))\}$. For each entry, the key is the term, and the value is a pair of the document along with the term's weight in this document. We then build a index m' separately for this document using a reduce function f. The reduce function is similar to allowing for a complementary function, but operates on a sequence of values instead of combining two values. For a term that appears in multiple corresponding documents, the reduce function f will organize them in the inner map (M_I), their values accumulated (+ as the combine function). Finally, we take a *union* of m with m'. When a term already exists in the current index, the value (inner indexes) are combined using a *union*. It can be written in code as follows.

	n	1 Core		72* Cores		Speed-up
	($\times 10^9$)	Time (secs)	Melts /sec	Time (secs)	Gelts /sec	
Build	1.96	1038	1.89	12.6	0.156	82.3
Queries	177	368	480.98	4.74	37.34	77.6

Table 13.1: The running time and rates for building and queering an inverted index – Here "one core" reports the sequential running time and "72* cores" means on all 72 cores with hyperthreads (i.e., 144 threads). Gelts/sec calculated as $n/(\text{time} \times 10^9)$.

$f(l = \{(d_i, w_i)\}) = M_I :: build\ (l, +)$

$m' = M_O :: build\ (\{(t_i, (d_i, w_i))\}, f)$

$m = M_O :: union\ (m, m', M_I :: union)$

13.2 Experiments

13.2.1 General Tests

To test the performance of the inverted index data structure described in Section 13.1, we use the publicly available Wikipedia database [271] (dumped on Oct. 1, 2016) consisting of 8.13 million documents. We removed all XML markup, treated everything other than alphanumeric characters as separators, and converted all upper case to lower case to make searches case-insensitive. This leaves 1.96 billion total words with 5.09 million unique words. We assigned a random weight to each word in each document (the values of the weights make no difference to the runtime). We measure the performance of building the index from an array of `(word, doc_id, weight)` triples, and the performance of queries that take an intersection (logical-and) followed by selecting the top 10 documents by weight.

Unfortunately we could not find a publicly available C++ version of inverted indices to compare to that support and/or queries and weights although there exist benchmarks supporting plain searching on a single word [110]. However the experiments do demonstrate speedup numbers, which are interesting in this application since it is the only one which does concurrent updates. In particular each query does its own intersection over the shared posting lists to create new lists (e.g., multiple users are searching at the same time). Timings are shown in Table 13.1. Our implementation can build the index for Wikipedia in 13 seconds, and can answer 100K queries with a total of close to 200 billion documents across the queries in under 5 seconds. It demonstrates that good speedup can be achieved for the concurrent updates in the query.

13.2.2 Concurrent Updates and Queries

We test "and"-queries, which means each query takes two terms and return the top-10 ranked documents in which both terms appear. We carefully choose the query terms such that the output is reasonably valid. The query is done by first read the posted-list of both terms, and take an *intersection* on them. Because of persistence the two posted-lists are just snapshots of the current database, and hence each query will not affect any other queries nor the update by the writer. We will show that simultaneous updates and queries does not have much overhead comparing to running them separately.

We first build a tree with 1.6×10^9 word-doc pairs. We use different number of threads to generate queries, and the rest are used for doing updates. We note that the thread allocation for running query/update ratio depends on the scheduler. We run both update and query simultaneously in 30 seconds, and record the throughput for each. We then test the same number of updates or queries running separately using all available threads (144 of them). Both update and query run in parallel—not only multiple queries run in parallel, but each single query is also parallel (using parallel intersection algorithm). The update uses a parallel union algorithm. We report the time for running them separately as T_u (purely update) and T_q (purely query). Numbers are shown in Table 13.2. As we use more threads to generate queries, the update ratio gets lower. This is because the sub-tasks in queries are generated more frequently, hence is more likely to be stolen. In conclusion, the total time of running them almost add up to 30 seconds, which is the time running them in parallel together.

In practice, the ratio of queries running on such search engines should be much more than the updates. In this case, our experiments show that adding a single writer to update the database does not cause much overhead in running time, and the queries and gradually get the newly-added documents.

p	T_u	T_q	$T_u + T_q$	T_{u+q}
10	13.4	17.3	30.7	30
20	8.22	21.6	29.82	30
40	4.18	25.1	29.28	30
80	1.82	27	28.82	30

Table 13.2: The running time (seconds) on the inverted index application – T_{u+q} denote the time for conducting updates and queries simultaneously, using p threads generating queries. We set T_{u+q} to be 30s. We then record the number of updates and queries finished running, and test the same number of updates/queries separately on the initial corpus. When testing separately we use all 144 threads.

```
struct inv_index {
  using word = char*;
  using doc_id = int;
  using weight = float;

  struct a_max {
    using aug_t = weight;
    static aug_t identity() {return 0;}
    static aug_t base(doc_id k,weight v) { return v;}
    static aug_t combine(aug_t a, aug_t b) { return (b ⟩ a) ? b : a;} };

  using post_elt = pair⟨doc_id, weight⟩;
  using post_list = aug_map⟨doc_id,weight,a_max⟩;
  using index_elt = pair⟨word, post_elt⟩;
  using index = map⟨word,post_list,str_less⟩;
  index idx;

  inv_index(index_elt* start, index_elt* end) {
    auto plist= [] (post_elt* s, post_elt* e) { return post_list(s,e,add); };
    idx.build_reduce⟨post_elt⟩(start,end,plist);}

  post_list get_list(word w) {
    maybe⟨post_list⟩ p = idx.find(w);
    if (p) return *p;
    else return post_list();  }

  post_list And(post_list a, post_list b) { return map_intersect(a,b,add);}

  post_list Or(post_list a, post_list b) { return map_union(a,b,add);}

  post_list And_Not(post_list a, post_list b) { return map_diff(a,b);}

  vector⟨post_elt⟩ top_k(post_list a, int k) {
    int l = min⟨int⟩(k,a.size());
    vector⟨post_elt⟩ vec(l);
    post_list b = a;
    for (int i=0; i ⟨ l; i++) {
      weight m = a.aug_val();
      auto f = [m] (weight v) {return v < m;};
      vec[i] = *b.aug_find(f);
      b.remove(vec[i].first); }
    return vec; } };
```

Figure 13.1: The data structure and construction of an inverted index, and the query operations.

Chapter 14

Write-Efficient Trees

Emerging non-volatile main memory (NVRAM) technologies, such as Intel's Optane DC Persistent Memory, are readily available on the market, and provide byte-addressability, low idle power, and improved memory-density. Due to these advantages, NVRAMs are likely to be the dominant main memories in the near future, or at least be a key component in the memory hierarchy. However, a significant programming challenge arises due to an underlying asymmetry between reads and writes—reads are much cheaper than writes in terms of both latency and throughput. Because bits are stored in these technologies as at rest "states" of the given material that can be quickly read but require physical change to update, this asymmetry is fundamental. This property requires researchers to rethink the design of algorithms and software, and optimize the existing ones accordingly to reduce the writes. Such algorithms are referred to as the ***write-efficient algorithms*** [145].

Blelloch et al. [46, 73, 75] formally defined and analyzed several sequential and parallel computation models with good caching and scheduling guarantees. The models abstract such asymmetry between reads and writes, and can be used to analyze algorithms on future memory. The basic model, which is the Asymmetric RAM (ARAM), extends the well-known external-memory model [16] and parameterizes the asymmetry using ω, which corresponds to the cost of a write relative to a read to the non-volatile main memory. The cost of an algorithm on the ARAM, the **asymmetric I/O cost**, is the number of write transfers to the main memory multiplied by ω, plus the number of read transfers. This model captures different system consideration (latency, bandwidth, or energy) by simply plugging in different values of ω, and also allows algorithms to be analyzed theoretically. Based on this idea, many interesting algorithms (and lower bounds) are designed and analyzed by various recent papers [46, 49, 70, 73, 75, 79, 80, 168].

In this thesis, we consider write-efficient algorithms for **join-based balanced binary search trees**, both theoretically and practically. Experimentally, we count the number of reads and write to main memory while simulating a variety of cache configurations. For I/O-bounded algorithms, we believe that the numbers are reasonable proxies for both running time (especially when implemented in parallel) and energy consumption.[1] Moreover, conclusions drawn from these numbers can likely give insights into tradeoffs between reads and writes among different algorithms.

To get these numbers, we use a framework based on a software simulator that can efficiently and precisely measure the number of read and write transfers of an algorithm. We also note that designing write-efficient algorithms falls in a high dimensional parameter space since the asymmetries on latency, bandwidth, and energy consumption between reads and writes are different. Here we abstract this as a single value ω. This value together with the cache size M and cache-line size B (set to be 64 bytes) form the parameter space of an algorithm.

14.1 Model and Simulator

To start with, we discuss how to measure the performance of algorithms on asymmetric memories. We begin with the computational model that estimates the cost of an algorithm. This model requires the numbers of read and write transfers between the non-volatile memory and the cache, so later we introduce how the numbers of an algorithm can be simulated. Unlike the existing symmetric memories, a simple cache policy like LRU does not work on some asymmetric settings. Thus in Section 14.1.2 we briefly summarize the solutions to fix it, and then the cache simulator given in Section 14.1.3 captures this number with different cache policies.

14.1.1 The Cost Model for Asymmetric Memory

The most commonly-used cost measure of an algorithm is the time complexity based on the RAM model, which is the overall number of instructions and memory accesses executed in this algorithm. Nowadays, since the actual latency of an access to the main memory is at least two orders of magnitudes more expensive than a CPU instruction, the *I/O cost* based on the external-memory model [16] is widely used to analyze the cost of an I/O-bounded algorithm. This 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), and it takes unit cost to transfer a single block between the small-memory and the large-memory. This cost measure estimates the running time reasonably well for I/O-bounded algorithms, especially in multi-core parallelism. An efficient algorithm in practice should achieve optimality in both the time complexity and the I/O cost.

To account for more expensive writes on future memories, here we adopt the idea of an (M, ω)-Asymmetric RAM (ARAM) [75]: similar to the external-memory model, transferring a block from large-memory to small-memory takes unit cost; on the other

[1] The energy consumption of main memory is a key concern since it costs 25-50% energy on data centers and servers [187, 192, 197].

direction, the cost is either 0 if this block is clean and never modified, or $\omega \gg 1$ otherwise. The **asymmetric I/O cost** Q[2] of an algorithm is the overall costs for all memory transfers.

14.1.2 Cache Policies

Either the classic external-memory model or the new ARAM assumes that we can explicitly manipulate the cache in the algorithm. This largely simplifies the analysis, and in many cases is provably within a constant factor of a more realistic cache's performance. For example, the standard least-recent used (LRU) policy is 2-competitive against the optimal offline cache-replacement sequence.

Interestingly, the competitive ratio does not hold in the asymmetric setting. Consider a cache with $k = M/B$ cache-lines and a memory access pattern that repeatedly writes to $k - 1$ cache-lines and read from other $k - 1$ cache-lines. An ideal cache policy will keep all $k - 1$ cache-lines associated to writes, so the I/O cost of each round is $k - 1$ for $k - 1$ read misses. An LRU policy however causes a cache miss for every single memory access, leading the I/O cost of each round to $\omega(k - 1) + k - 1$. This overhead is proportional to ω, which can be significant and problematic.

The solution is affected by the architecture, depending on whether software explicitly controls a DRAM buffer or not [105, 115, 186, 235]. If so, then the cost measures on the these models are just the costs in practice, but programmers are responsible for managing what to put on the small-memory and guaranteeing correctness. The other option is to leave the hardware to control the small-memory. In this case, Blelloch et al. [73] show that if the small-memory is partitioned into two equal-size pools and each of them is maintained using LRU policy, the performance is 3-competitive against the optimal offline cache-replacement sequence (e.g. using 3× space and incurring no more than 3× cost).

We consider three different cache policies in this chapter. In this thesis, we only use the policy maintains the small-memory as one memory pool and uses the LRU policy for replacement. Other policies and applications can be found in [147].

14.1.3 The Cache Simulator

To capture the number of reads and writes to the main memory, we developed a software simulator. The cache simulator is composed of an ordered map that keeps tracks of the time stamp of the last visit to each cache-line in the current cache, and an unordered map that stores the mapping from each cache-line to the corresponding location in the ordered map if this cache-line is currently in the cache.

The cache simulator encapsulates a new structure Array that is used in coding algorithms in this chapter. It is like a regular array that can be dynamically allocated and freed, and supports two functions: Read and Write to a specific location in this array. The Arrays are responsible for reporting the memory accesses of the algorithm to the

[2] Throughout the chapter, we abbreviate it as the *I/O cost*, unless stated otherwise explicitly.

cache simulator, and the cache simulator will update the state of the cache accordingly. Therefore, coding using the Arrays is not different from regular programming much.

The memory accesses to loop variables and temporary variables are ignored, as well as the call stack. This is because the number of such variables is small in all of the algorithms in this chapter (usually no more than 10). Meanwhile, the call stack of all algorithms in this chapter has size $O(\log n)$. The overall amount of uncaptured space is orders of magnitudes smaller than the amount of fast memory in our experiments.

The cache consists of one memory pool. The cache simulator maintains two counters in the memory pool: the number of **read transfers**, and the number of **write transfers**. When testing each algorithm on a specific input instance, the cache is emptied at the beginning and flushed at the end. A read or write is free if the location is already in the cache; otherwise the corresponding cache-line is loaded, the counter of read transfer increments by 1, and the least-recently-used cache-line in this pool is evicted. Also, a write will mark the dirty-bit of the cache-line to be `true`. When evicting a dirty cache-line, the counter of write transfer increments by 1. Notice that memory reads can cause write transfers, and memory writes can lead to read transfers.

14.2 Write-Efficient Tree Algorithms

We now study the write-efficiency of binary tree algorithms. In this thesis, we consider AVL trees, red-black trees, weight-balanced trees, and treaps. Weight-balanced trees are inherently write-inefficient since any update requires tree size changes to every involved tree nodes. We first analyze the I/O cost of the other three trees for single updates, and show experiment results.

14.2.1 I/O cost on BSTs

For simplicity, here we assume that the small-memory size is $M = O(1)$. Locating the key for a lookup, insertion or deletion requires to load and compare to $\Theta(\log n)$ tree nodes on the balanced binary search trees (BSTs), The I/O cost is $\Theta(\log n)$, which is also the lower bound of such operations.

For an insertion or deletion, we also need to modify the tree accordingly. In the asymmetric setting, weight-balanced trees [216] are not a good option since we have to update the subtree sizes all the way to the root. This update leads to $\Theta(\log n)$ writes to the large-memory per update. For the other types of BSTs, we show their I/O costs on insertions and deletions individually.

Red-black Trees. Red-black trees [41, 151] have the simplest update rules among these balanced BSTs. With the classic rebalancing rules and careful implementation, it requires only $O(1)$ amortized time per update (insertion or deletion) after locating the key [265]. Also, red-black trees require no extra cost to update balancing information except for the tree nodes involved in rotations (unlike the case in AVL trees). As a result, red-black trees have an optimal amortized I/O cost $Q = \Theta(\log n)$ per a lookup and $Q = \Theta(\omega + \log n)$ per insertion or deletion on the (M, ω)-ARAM.

AVL Trees. An insertion in AVL trees requires at most two rotations (a double rotation) [13]. Unlike the red-black trees however, we need to track and update the balance factors along the path from the root to the modified tree node. We now bound the number of updated balance factors to be a constant. If we store the height of the subtree in each tree node, the difference of the two subtree heights can be checked in constant time. Since the number of subtrees of height d in an AVL tree is no more than $n/c^{\lfloor d/2 \rfloor}$ for a tree with n nodes and some constant $c > 1$ (Section 3.3), the number of increments of the counts for n nodes is $\sum_{d \geq 1} d \cdot (n/c^{\lfloor d/2 \rfloor}) = O(n)$. On average, an insertion needs $O(n)/n = O(1)$ writes.

The deletions of AVL trees is more complicated and $\Theta(\log n)$ rotations can be applied on every deletion. Amani et al. [24] recently showed that there exists such a sequence of $3n$ intermixed insertions and deletions on an initially empty AVL tree that takes $\Theta(n \log n)$ rotations. This instance indicates that the classic implementation of AVL trees has an I/O cost $Q = \Theta(\omega \log n)$ per deletion in the worst case.

Treaps. A treap, also called as a randomized search tree [246], is a Cartesian tree in which each key is given a randomly chosen numeric priority, and the inorder traversal order of the nodes is the same as the sorted order of the keys. The priority for any non-leaf node must be greater than or equal to the priority of its children.

When inserting an element into a treap with $n - 1$ elements or removing an element from n elements, The updated element only compares to the elements that each has a higher priority than the other elements between this element and the updated element. The number of such comparisons is $\sum_{j \in [n] \setminus \{i\}} 1/|i - j| = O(\log n)$ in expectation[3], where i is the position of the updated element in the total order of n elements [246].

The number of rotations can be computed similarly. For an insertion, a rotation happens once the inserted element has a higher priority than all elements in the entire subtree. Again if we assume the inserted element ranked i-th in the total order, the probability that it rotates up for the j-th ranked tree node is $1/|i - j|^2$ (i.e., the j-th element has higher priority than all elements between, and lower than the priority of the inserted node). The overall expected number of rotations per insertion is $\sum_{j \in [n] \setminus \{i\}} 1/|i - j|^2 = O(1)$ in expectation. We can show the constant writes per deletion accordingly.

We note that unlike an AVL tree or a red-black tree, a treap does not require updates to the balancing criteria, which means that we never need to modify the information in each tree node after it is inserted. As a result, an insertion, deletion or query on a treap requires $O(\log n)$ reads *whp* and an insertion or deletion requires $O(1)$ writes in expectation.

[3] Also with high probability.

14.2.2 Implementation

We implement theses algorithms using *join*-based algorithms. We note that, *join* is the only function that *writes* to the memory. More specifically, all operations that change the attributes of a tree node (e.g., linking to new children, height-maintaining for AVL or red-black trees, color-changing in red-black trees, etc.) are restricted in *join*. This property also greatly simplifies the counting of writes in our simulator and makes the optimizations to reduce writes easier.

There are several benefits of using the *join*-based framework for our implementation and experiment. First, as we just explained, different updates have the uniform code on different types of BSTs (except for the Join), which justifies the performance by the different balancing criteria for the BSTs, instead of the different implementations for different trees. Second, although the joined-based implementation operates on two trees, one can check that when one tree contains only a singleton element, the algorithm runs the same as the algorithm of a single insertion on each type of the BSTs. The deletion can also be implemented by taking the difference by the original tree and a tree with a single element. As a result, the joined-based implementation is strictly more powerful. Lastly, we can also run interesting experiments on more operations like bulk updates, and compare the results on different BSTs.

14.3 Experiments

We now show the experimental results on the counts of read/write transfers for different settings. Due to the page limit, our experiment mainly focuses on the performance of various binary search trees (AVL trees, red-black trees, and treaps) with different batch sizes.

In the experiment, we first insert $m = 10^6$ million integers as keys to a tree T (empty at the beginning), drawn from a uniform distribution from 32-bit unsigned integers, and then delete them in a uniformly random order. The insertion and deletion are grouped in batches of size s, indicating that the insertions are $\lceil m/s \rceil$ unions on the main tree T with trees of size s. The deletions are also batched in $\lceil m/s \rceil$ bulks of size s. In our experiment, we construct a smaller tree for each batch and then call the union or difference function we just mentioned. We note that if all update elements are given in advance, we can also sort them and put them in a list. Since this is a more specific case, our experiment is based on the tree-tree updates.

The node size in all different types of tress is 16 bytes. Each tree node stores four 4-byte data blocks to hold the key, the left and right pointers, and the balancing information. The cache contains 10,000 cache-lines, similar to the setting for unordered sets.

Table 14.1 shows the experimental results on BSTs with different balancing schemes with various batch sizes. The numbers are read and write transfers per update.

(a) Insertion / Union

Batch	AVL		RB-Tree		Treap		$\omega = 10$			$\omega = 100$		
Size	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**AVL**	**RB-T**	**Treap**	**AVL**	**RB-T**	**Treap**
1	11.28	2.83	12.00	3.48	16.61	1.79	39.6	46.8	34.5	295	360	196
1k	12.67	2.89	13.29	3.66	17.18	1.89	41.6	49.9	36.1	302	379	206
10k	6.18	2.68	6.40	3.01	7.33	1.85	33.0	36.5	25.8	274	308	192
100k	2.54	1.84	2.54	1.86	2.56	1.66	20.9	21.2	19.2	186	189	169

(b) Deletion / Difference

Batch	AVL		RB-Tree		Treap		$\omega = 10$			$\omega = 100$		
Size	**RT**	**WT**	**RT**	**WT**	**RT**	**WT**	**AVL**	**RB-T**	**Treap**	**AVL**	**RB-T**	**Treap**
1	13.83	2.72	16.17	5.17	17.85	1.98	41.1	67.8	37.7	286	533	216
1k	15.11	2.79	17.65	5.21	18.52	2.08	43.0	69.8	39.3	294	539	227
10k	8.17	2.69	10.43	3.44	9.09	2.06	35.1	44.9	29.7	278	355	215
100k	3.22	1.99	3.54	2.43	3.23	1.78	23.2	27.8	21.1	203	246	182

(c) Average Tree Depth

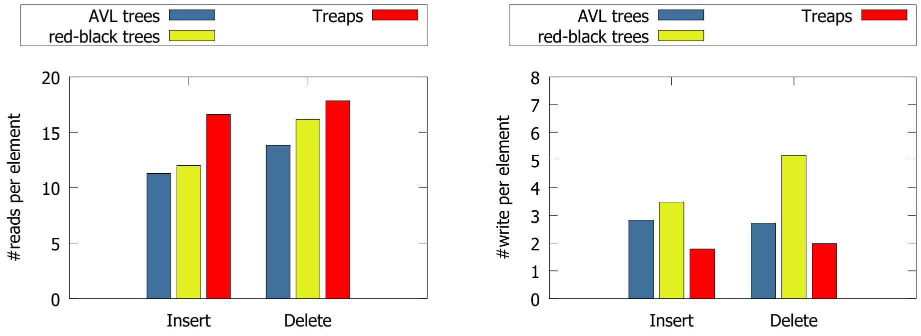
AVL	RB-Tree	Treap
19.39	19.62	26.48

Table 14.1: Numbers of read and write transfers and asymmetric I/O costs of different BSTs with various batch sizes – The numbers are divided by 10^6 (i.e., per inserted/deleted elements). The write-read ratio ω are selected to be typical projected values 10 (latency, bandwidth) and 100 (energy).

Batch Size 1. We first look at the case corresponding the single insertions and deletions, and the results are shown in Figure 14.1 and 14.2. Regarding the number of writes, treaps show the best performance. This is easy to understand since treaps do not modify any information for rebalancing during insertions/deletions. The structure of treaps is deterministic once the priorities are decided. The priorities are set before the merging (deleting), and never changed later. For such reasons, treaps require much fewer writes per update compared to AVL and red-black trees.

The AVL and red-black trees maintain the balancing information on each tree node that needs to be updated when the subtree height changes. Then more writes are used due to these updates. Between these two types of BSTs, red-black trees require more writes, since there are also some color flips on the siblings of the updated tree path. Such flips are extremely cheap in the classic symmetric setting but will cost much in the asymmetric setting when writes become expensive.

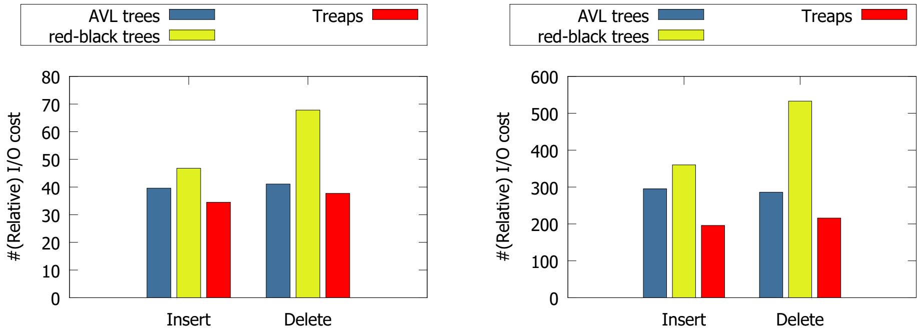
The fewer writes for treaps come together with the extra cost on more reads. Treaps are less strictly balanced compared to the other two trees, which saves the writes to



(a) Number of reads

(b) Number of writes

Figure 14.1: The number of read and write transfers of different BSTs on single insertion/deletion – Data are from the first rows in Table 14.1(a) and 14.1(b).



(a) $\omega = 10$

(b) $\omega = 100$

Figure 14.2: The I/O cost of different BSTs on single insertion/deletion – Data are from the first rows in Table 14.1(a) and 14.1(b).

maintaining such balancing, but leads to larger average tree depth (shown in Table 14.1(c), about 30% deeper than the other trees). The average depths for AVL and red-black trees are close to optimal, which is 18.96 for a perfectly balanced tree with 10^6 nodes. Because of the shallower average depth, the number of reads required in either updates or queries is much small on these two trees.

Larger Batch Sizes. We now discuss how does the batch size affect the numbers of read and write transfers. The numbers are given in Table 14.1.

When the batch size increases but remains small, the reads and writes almost remain the same and slightly increase. When the batch size is smaller, the elements in a batch and the paths to visit them are always loaded into the cache once and always stay there. Therefore, the different batch sizes less than 100 do not affect the performance much.

The peak I/O cost per update is around batch size 1000, as we show here. In such case, the overall footprint of a batch update no longer fit into the cache, which may lead to two loads per tree node (once in the split phase and the other in the join phase).

However, the cost turns down when the batch size grows over 1000. The reason is that, the number of tree nodes visited during each bulk update is $\Theta(s\log(m/s))$ (recall that s is the bulk size), which is also the time complexity [74] of this process. Namely, we need to visit $O(\log(m/s))$ tree nodes per inserted node, so we touch fewer nodes as s goes up. Compared to the single updates, each node on multiple tree paths[4] is only looked at and compared with once in the joined-based bulk updates, and this is from which the improvement comes. Since the top levels in the trees are visited more frequently, they usually stay in the cache. As a result, the saved memory accesses for small batch size are hidden by the function of the cache. However, when the batch size keeps growing and exceed the cache size, then the saved memory accesses lead to lower reads, as shown in Table 14.1.

The number of writes also decreases for larger batch sizes in AVL and red-black trees, because of the previously stated reason. When the elements are inserted or deleted one by one, the balancing factor of a node on by multiple tree paths can be updated multiple times. On the other hand, when the tree is updated in a bulk, such information will be updated by at most once at the join point, which saves the number of writes to the asymmetric memory. However, since treaps do not need to update such information, we cannot observe a significant drop-off on writes for larger batch sizes.

Queries. We have not explicitly tested the I/O costs for queries, since most queries (like finding or checking a key, locating the k-th element) have the same memory access pattern as the insertions. The only difference is that they do not modify the tree, so there will be no writes. These updates may flush the dirty cache lines, and thus slightly increase the writes. Our experiment shows that if we have the same number of queries and insertions, the number of writes increases by no more than 5%, which is insignificant. Therefore, we believe that we can ignore such changes in the most cases.

14.4 Conclusions

In this section, we theoretically analyze the asymmetric I/O costs of different types of binary search trees. We show that red-black trees, the insertions for AVL trees, and treaps on expectation have an optimal asymptotic cost ($\Theta(\omega+\log n)$ per update).

We then test the actual performance by conducting experiments based on the join-based implementation, and show that treaps have the best update cost in most cases. The advantage comes from a looser balancing constraint, which also leads to a larger tree depth and query costs. As a result, AVL tree will be a better option if the queries are much more than the updates.

[4] Usually on the top part of the tree. For example, every single insertion visits the root node.

Part III

Related Work, Conclusion and Future Work

Chapter 15

Related Work

15.1 Join-based Algorithms

Tarjan first studied the *join* and *split* functions in [265]. Tarjan showed how to efficiently implement the functions on red-black and splay trees but did not give any applications for the two functions. Adams [10, 11] applied *join* and *split* on weight balanced trees to some bulk functions and showed an elegant way to implement *union*, *intersection* and *difference*. The method was then implemented in some languages and libraries such as MIT/GNU Scheme in Haskell [200]. Adams' paper was reported buggy [159, 257] in parameter choosing and also in that the *join* (which is called *concat3* in his paper) does not rebalance the tree, but the framework of set functions implementation is still used and studied today. Adams did not consider parallelism in the algorithm, but the divide-and-conquer scheme is inherently parallel.

join and *split* appear in the LEDA library [209] for sorted sequences, and the CGAL library for ordered maps [270]. None of this work considered parallel algorithms based on the functions, nor how to build an interface out of just *join*. Frias and Singler [135] use *join* and *split* on red-black trees for an implementation of the MCSTL, a multi-core version of the C++ Standard Template Library (STL). Their algorithms are lower level based on partitioning across processors, and are for bulk insertion and deletion. The functions *join*, *split* and *join2* are also studied and used in various previous work for trees to support multiple applications [51, 108, 246].

15.2 Set-set Algorithms

Merging two ordered sets has been well studied in the sequential setting. Hwang and Lin [164] describe an algorithm to merge two arrays, which costs optimal work. Their algorithm works for arrays, and since writing back both array costs $O(m + n)$ work, the algorithm only returns the cross pointers between two arrays. Brown and Tarjan [91] considered input data to be arranged in a BST, which allows the merged result to be explicitly given by a new BST in time $O\big(m\log(\frac{n}{m}+1)\big)$. Their algorithm works on AVL and 2-3 trees. None of these algorithms considered parallelism. Katajainen et al. [176] studied the space-efficiency on merging two sets in parallel. Their focus is not on reducing time complexity. Furthermore the above-mentioned works are not based on join and are much more complicated than our algorithm.

There is also previous work studying parallel set operations on two ordered sets, but each previous algorithm only works on one type of balance tree. Paul, Vishkin, and Wagener studied bulk insertion and deletion on 2-3 trees in the PRAM model [228]. Park and Park showed similar results for red-black trees [227]. These algorithms are not based on *join* and are not work efficient, requiring $O(m\log n)$ work. Katajainen [175] claimed an algorithm with $O\big(m\log(\frac{n}{m}+1)\big)$ work and $O(\log n)$ span using 2-3 trees, but it appears to contain some bugs in the analysis [71]. 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 [22] 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 thesis is in showing that very simple algorithms are work efficient and have polylogarithmic span, and less with optimizing the span.

Many researchers have considered concurrent implementations of balanced search trees (e.g., [90, 182, 184, 213]). 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.

Researchers have also studied distributed memory implementations of maps and sets, including a distributed version of STL as part of the HPC++ effort [170], and the STAPL library [264]. The emphasis of this work is on how the maps and sets are partitioned across the memories.

15.3 Computational Geometry

Many data structures are designed for solving range, segment and rectangle queries, such as range trees [56], segment trees [58], kd-trees [54], R-trees [44, 240, 245], priority trees [204], and many others [80, 118, 199, 222, 244, 272]. They are then applied to a variety of other problems [15, 20, 21, 27, 29, 57, 67, 68, 89, 140, 167, 225].

There have been fundamental sequential data structures for such queries. The classic range tree [56] has construction time $O(n\log n)$ and query time $O(k+\log^2 n)$ for input size n and output size k. Using fractional cascading [103], the query time can be $O(k+\log n)$. We did not employ it for the simplicity and extensibility in engineering, and we believe the performance is still efficient and competitive. The terminology "segment tree" [58, 101] refers to different data structures in the literature, but the common insight is to use the tree structure to partition the interval hierarchically. Our version is similar to some previous results [17, 32, 101]. Previous solutions for rectangle queries usually use combinations of range trees, segment trees, interval trees, and priority trees [58, 106, 130, 131]. Some other prior work focused on developing sequential sweepline algorithms for range queries [28, 31], segment intersecting [102, 208] and rectangle queries [202]. There are also sequential I/O-efficient algorithms for computational geometry [27, 29, 107] and sequential libraries support such queries [5, 98].

In the parallel setting, there exist many theoretical results [14, 17, 33, 34, 35, 139, 141]. Atallah et al. [34] proposed the *array-of-trees* using persistent binary search trees (BST) to store all intermediate trees in a sweepline algorithm by storing all versions of a tree node in a super-node. Our method also uses persistent BSTs, but uses path-copying to maintain a set of trees independently instead of in one skeleton. Recently, Afshani et al. [14] implemented the array-of-trees for the 1D total visibility-index problem. Atallah et al. [35] discussed a cascading divide-and-conquer scheme for solving computational geometry problems in parallel. Goodrich et al. [141] proposed a framework to parallelize several sweepline-based algorithms. There has been previous work focusing on parallelizing segment-tree-like data structures [17, 32], and our segment tree algorithm is inspired by them. There are also theoretical I/O efficient algorithms in parallel [21, 253]. We know of no implementations or experimental evaluations of these theoretically efficient algorithms on range, segment and rectangle queries. There are also parallel implementation-based works such as parallel R-trees [172], parallel sweepline algorithms [205], and algorithms focusing on distributed systems [275] and GPUs [274]. No theoretical guarantees are provided in these papers.

15.4 Range Queries

Many researchers have studied concurrent algorithms and implementations of maps based on balanced search trees focusing on insertion, deletion and search [39, 66, 90, 95, 126, 132, 135, 182, 184, 190, 213]. Motivated by applications in data analysis recently researchers have considered mixing atomic range scanning with concurrent updates (insertion and deletion) [40, 94, 234]. None of them, however, has considered sub-linear time range sums.

There has also been significant work on parallel algorithms and implementations of bulk operations on ordered maps and sets [22, 71, 74, 125, 132, 135, 165, 227, 228]. Union and intersection, for example, are available as part of the multicore version of the C++ Standard Template Library [135]. Again none of this work has considered fast range sums. There has been some work on parallel data structures for specific applications of range sums such as range trees [179].

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. [199]). Several of these approaches have been implemented as part of systems [180, 215]. 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 4.4.

Various forms of range queries have been considered in the context of relational databases [99, 137, 144, 152, 161]. Ho et. al. [161] 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 [15, 23]. 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 [25, 191, 221, 239] support fast range sums.

15.5 Database Management Systems

MVCC and Snapshot Isolation. Concurrency control is an important issue for DBMS design, and has been studied for decades. One classic way is to rely on the *Two-Phase Locking (2PL)* [269]. Although 2PL guarantees serializability, it causes readers and writers to block each other. In fact, any lock-based isolation mechanism has the same issue, which will be particular inefficient in a scenario with heavy-loaded read-only transactions. MVCC [62, 211, 269] is a widely-used technique in various DBMSs for allowing fast and correct read transactions without blocking (or being blocked by) the writers. MVCC is implemented in many DBMSs [123, 133, 185, 214, 233, 251]. Some of these existing systems, like Hekaton [185] and HyPer [214], use timestamp-based version chain for concurrency control. This may require the read transactions to scan the version chain and check the visibility of the version at the current timestamp, which can be expensive when there are a large number of versions. HyPer uses the *version synopses* to avoid expensive scan, but this does not have any theoretical guarantee in bounding the reading time. P-Trees avoid this by using the functional data structure.

Copy-on-write and Functional Data Structures. Copy-on-write (CoW) means to explicitly copy some resource if and only if it is modified. Indeed path-copying is a specific implementation of CoW. Similar ideas are also used in *shadow paging* [194] to guarantee atomicity. Path-copying has been used in maintaining multiversion B-tree or B+tree structures or their variants [43, 255]. Path copying is the default implementation in functional languages, where data cannot be overwritten [220]. It is widely-used in real-world database systems for version-controlling like LMDB [4], CouchDB [26], Hyder [63] and InnoDB [136], as well as many other systems [86, 109, 120, 160, 174]. Most of these only allow a single update at a time, sequentializing the writes. Hyder supports merging of path copied trees. This means that transactions can update trees concurrently, and then each is merged into the main trees one at a time. This allows for some pipelining [64] but still fully sequentializes the commits at the root (only one merge can complete at a time). It also means that transactions can abort during the merge if they have any conflicts. Importantly these other systems have focused on point updates (single insertions, deletions, or value changes). An important aspect of P-Trees is that they support parallel bulk updates on pure trees. This allows for batching of transactions in certain situations and processing them in parallel to get much higher throughput, as shown in the YCSB benchmark. Existing work mostly focusing on single-operation OLTP primitives insertion, deletion and lookup. Our work differs from them in several aspects.

1. To serialize writes, some of them sequentialize all updates [4]. Hyder uses melding to merge unserialized versions. Although a pipeline scheme is proposed to accelerate the melding algorithm, the process is still inherently sequential, which is a major bottleneck of the system. P-Trees support parallel bulk operation using divide-and-conquer for better exploit parallelism.
2. As mentioned in Section 11.2, P-Trees copy asymptotically less tree nodes than existing systems when committing a batch of updates.
3. Hyder aborts transactions if there is unresolvable conflict, while P-Tree uses batching and never aborts.
4. Some existing solutions uses CoW on B-trees [4, 255], which is generally expensive, because tree node are usually large. P-Trees are binary, and thus each tree node stores one single tuple, making copying fast. Especially for maintaining in-memory DBMSs, one cache line could only hold one or two tree nodes of P-Trees, which makes our binary tree have competitive performance to B-trees in most of the cases.
5. Existing CoW systems are optimized for OLTP setting, and there is few evaluations of them on interesting OLAP benchmarks such as TPC-H. P-Trees also outperforms existing systems optimized for OLAP queries on TPC-H.
6. P-Tree relies on an efficient batching algorithm. When update transactions are invoked concurrently, the batching process also costs overhead and long latency, which is similar to Hyder's melding [238] and Calvin's sequencer [266].

Query Optimizations. Our proposed index nesting is related to many previous techniques proposed to support high throughput of OLAP queries. As mentioned, the paired index enables virtually denormalization [162] of two tables. The main challenging of physical denormalization is the high memory assumption, and the high cost to update while maintaining SI. There are attempts aiming at reducing the space required by data denormalization using compression [193]. P-Trees avoid these costs because we never physically copy data multiple times by making use of the nested trees.

Although our nested indexes usually provide a view of pre-join over tables, our approach differs from materialized views [138] in that we do not physically create a new table, such that the update is still cheap using path-copying.

Table partitioning means to partition a table in storage to represent the parent-child relation [19, 37, 128]. Table partitioning was also previously employed to manage TPC-H workload, especially to manage the belongness between orders and lineitems. Our index nesting conceptually uses the same idea of putting all lineitems in the same order together, but differs in that we build an index using a tree inside. Also, these previous work on table partitioning focusing more on partitioning the table for distributed systems. The idea of representing relations between objects across tables in a graphical or hierarchical way is similar to the Resource Description Framework (RDF) [36, 87, 223] and path-indexing [65, 196] in Object-Oriented Databases (OOD). Our index nesting differs from them in that we usually only allow for regular tree or DAG nesting, instead of arbitrary graphic relations. Also, we propose to use nested pure tree structures to support fast queries and updates on such indexes.

Chapter 16

Future Work

Batching Algorithms. In this thesis, we apply P-Trees to obtain a concurrent data structure allowing for serializability, lock-free updates and wait-free read transactions. Especially to enable serializability, this thesis uses batching to collect concurrent updates and commit them in one big transaction. The current batching strategy, however, is simple and does not have any theoretical guarantee. To achieve reasonable tradeoff between latency and throughput, the batching algorithm must consider the parameters of the system. One interesting future direction is to conduct in-depth study of a better batching algorithm. The problem can be roughly described as given the throughput of the system, the arriving rate of new operations, the cost function $f(x)$ of committed a batch of x elements, and the current waiting elements in the pool, deciding the proper time to commit a batch of y elements to minimize the mean latency of each operation.

The goal can be either a thorough theoretical study or an experimental study of this problem. Theoretically, we are interested in looking at the preferred algorithms and conditions to make the system stable, and even to achieve the lowest mean latency. It is also possible to study this from a pure system perspective, which means to design batching algorithms to achieve high performance for different platforms.

Inverted Index Searching with Version Maintenance and Compression. In Section 13.1, this thesis studies the inverted indexes for document searching. Allowing concurrent update and queries to the database require the index the support multi-versioning. The current implementation does not consider garbage collection on such a system, but this is necessary and important in practice. Our inverted index can be combined with the VM problem as introduced in Chapter 12 or other VM solutions [50]. Recently, Dhulipala et al. [122] applied *join*-based framework to C-trees, which enables effective compression on trees. This is useful in developing large-scale in-memory database systems. It is of interest to apply both the VM problem and the compression on such a inverted index searching system.

Augmentation for OLAP DBMS. The work in this thesis builds DBMS for fast OLAP queries using P-Trees. However, the augmentation in P-Trees is not exploited in the current implementation. In fact, a lot of aggregation-based OLAP queries, such as some TPC-H ones, can be enhanced by using proper augmentations. On the other hand, using augmentation brings up extra overhead in space. It is worth studying the tradeoff and good strategy to use augmentations for fast OLAP queries.

Parallelize Concurrent Update Transactions in DBMS. The current tested HTAP workloads for our system is query-dominated, we thus sequentialize all update transactions to achieve serializability. In workloads that the updates are more frequent, it is helpful to also parallelize the updates. Although for simple point transactions, batching is effective and efficient, for large transactions, the dependency and conflict among them can be arbitrary and hard to resolve. Also, for certain pre-defined transaction types, such as the TPC-C transactions, the conflict is likely to be rare, and easier to detect. A interesting future direction is to study the parallelization of update transactions, both for a general and for a specific workload.

Set Algorithms with Optimal Depth. This thesis present an efficient algorithm that merges two ordered sets into another ordered structure, which is work-optimal ($O(m\log(\frac{n}{m}+1))$) for two sets with sizes n and $m \leq n$) and has span $O(\log n \log m)$. In practice, this algorithm achieves good parallelism, but theoretically, it is of interest to study if we can reduce the span to $O(\log n)$, while still achieve work-optimality. This, unfortunately, possibly requires to break up the *join*-based framework. Interestingly, in the binary-forking model, there is no previous algorithms that achieves both optimal work and span. Some initial result of studying this problem is presented in the unpublished work [81].

Experimental Study of Write-efficient Tree Algorithms. This thesis presents initial study of write-efficient tree algorithms motivated by the new NV-RAM hardware. The work was mostly done on a software simulator since the memory was not available when the study was conducted. More recently, the new hardware has been manufactured is going to be available in commodity. As a result, it is of importance to study the performance of the write-efficient algorithms and testing their actual wall-clock performance on the real machine.

More Balancing Schemes. This thesis proposes the *join*-based algorithmic framework, as well as a list of rule to make balancing schemes joinable. The current work is shown to be applicable to four well-known balancing schemes. I am interested in extending the current framework, and including more balancing schemes in the framework.

Chapter 17

Conclusion

This thesis studies efficient tree algorithms, both in theory and in practice, and also applies them to various applications.

One of the core contribution of this thesis is the *join*-based algorithmic framework. By abstracting out *join* from tree algorithms, one can get simple tree algorithms with a rich set of functionalities. Especially, *join* deals with all rebalancing, which makes all other algorithms as well as all analysis based on ranks that are generic across balancing schemes. *join* also captures all properties needed for augmentation and persistence, such that other algorithms are oblivious to these functionalities. As such, we are able to obtain both new theoretical results and efficient new implementations for some applications. The new theoretical results includes the work-efficient parallel set-set algorithms and its proof, the efficient computational geometry algorithms in the elegant augmented map framework, and parallel sweepline algorithms with low depth.

The other contribution of this thesis is the augmentation framework both for augmented trees and augmented maps. This provides elegant formalization to several seemingly-unrelated applications. This especially allows for concise implementation for all of them.

Also, based on the thesis work, I implemented P-Trees in the PAM library. The library enables the useful properties including parallelism, concurrency, persistence and precise GC. The library is open source. By applying the library, one can obtain simple and efficient implementations of many applications, include but not limited to those mentioned in this thesis.

Finally, this thesis presents novel and efficient implementation to a list of real-world applications. Most of these implementations are based on the P-Trees in PAM with very concise upper-level code. Based on the general-purpose library PAM, our implementations outperform several existing implementations that are specific to the applications. This demonstrate the advantage and practicality of P-Trees.

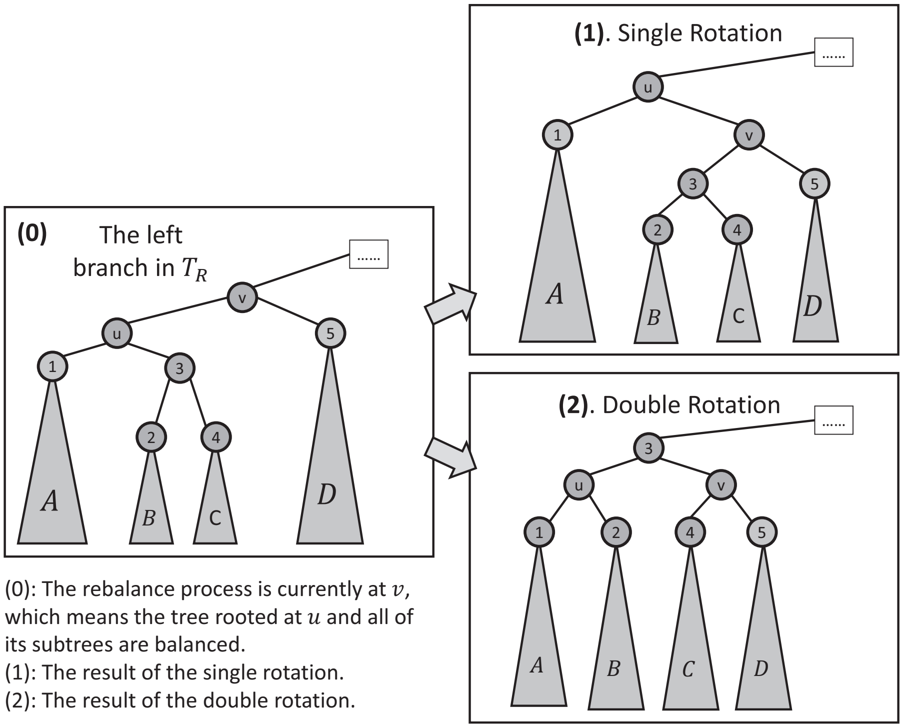
Part IV

Appendix

Appendix A

Proofs and Analysis

A.1 Proof for Lemma 3.2.6



(0): The rebalance process is currently at v, which means the tree rooted at u and all of its subtrees are balanced.
(1): The result of the single rotation.
(2): The result of the double rotation.

Figure A.1: An illustration of two kinds of outcomes of rotation after joining two weight balanced trees – After we append the smaller tree to the larger one and rebalance from that point upwards, we reach the case in (0), where u has been balanced, and the smaller tree has been part of it. Now we are balancing v, and two options are shown in (1) and (2). At least one of the two rotation will rebalance v.

Proof. Recall that in a weight balanced tree, for a certain node, neither of its children is β times larger than the other one, where $\beta = \frac{1}{\alpha} - 1$. When $\alpha \le 1 - \frac{1}{\sqrt{2}}$, we have $\beta \ge 1 + \sqrt{2}$.

WLOG, we prove the case when $|T_l| < |T_r|$, where T_l is inserted along the left branch of T_r. Then we rebalance the tree from the point of key k and go upwards. As shown in Figure A.1 (0), suppose the rebalance has been processed to u (then we can use reduction). Thus the subtree rooted at u is balanced, and T_l is part of it. We name the four trees from left to right A, B, C and D, and the number of nodes in them a, b, c and d. From the balance condition we know that A is balanced with $B + C$, and B is balanced to C, i.e.:

$$\frac{1}{\beta}(b+c) \le a \le \beta(b+c) \tag{A.1}$$

$$\frac{1}{\beta}b \le c \le \beta c \tag{A.2}$$

We claim that at least one of the two operations will rebalanced the tree rooted at v in Figure A.1 (0):

1. Single rotation: right rotation at u and v (as shown in Figure A.1 (1));
2. double rotation: Left rotation followed by a right rotation (as shown in Figure A.1 (2)).

Also, notice that the inbalance is caused by the insertion of a subtree at the leftmost branch. Suppose the size of the smaller tree is x, and the size of the original left child of v is y. Note that in the process of *join*, T_L is not concatenated with v. Instead, it goes down to deeper nodes. Also, note that the original subtree of size y is weight balanced with D. This means we have:

$$x < \frac{1}{\beta}(d+y)$$
$$\frac{1}{\beta}d \le y \le \beta d$$
$$x + y = a + b + c$$

From the above three inequalities we get $x < \frac{1}{\beta}d + d$, thus:

$$a + b + c = x + y < (1 + \beta + \frac{1}{\beta})d$$

Since a unbalance occurs, we have:

$$a + b + c > \beta d$$

We discuss the following 3 cases:

1. **$B + C$ is weight balanced with D**, i.e.,

$$\frac{1}{\beta}(b+c) \le d \le \beta(b+c) \tag{A.3}$$

In this case, we apply a right rotate. The new tree rooted at u is now balanced. A is naturally balanced.

Then we discuss in two cases:

(a) $\boldsymbol{\beta a \geq b + c + d}$.

Notice that $b + c \geq \frac{1}{\beta}a$, meaning that $b + c + d > \frac{1}{\beta}a$. Then in this case, A is balanced to $B + C + D$, $B + C$ is balanced to D. Thus just one right rotation will rebalance the tree rooted at u (Figure A.1 (1)).

(b) $\boldsymbol{\beta a < b + c + d}$.

In this case, we claim that a double rotation as shown in Figure A.1 (2) will rebalance the tree. Now we need to prove the balance of all the subtree pairs: A with B, C with D, and $A + B$ with $C + D$.

First notice that when $\beta a < b + c + d$, from (A.3) we can get:

$$\begin{aligned}
&\beta d < a + b + c < \frac{1}{\beta}(b + c + d) + b + c \\
\Rightarrow &(\beta - \frac{1}{\beta})d < (\frac{1}{\beta} + 1)(b + c) \\
\Rightarrow &(\beta - 1)d < b + c \qquad \text{(A.4)}
\end{aligned}$$

Considering (A.3), we have $(\beta - 1)d < b + c \leq \beta d$. Notice b and c satisfy (A.2), we have:

$$b > \frac{1}{\beta + 1}(b + c) > \frac{\beta - 1}{\beta + 1}d \qquad \text{(A.5)}$$

$$c > \frac{1}{\beta + 1}(b + c) > \frac{\beta - 1}{\beta + 1}d \qquad \text{(A.6)}$$

Also note that when $\beta > 1 + \sqrt{2} \approx 2.414$, we have

$$\frac{\beta + 1}{\beta - 1} < \beta \qquad \text{(A.7)}$$

We discuss the following three conditions of subtrees' balance:

i. **Prove $\boldsymbol{A}$ is weight balanced to $\boldsymbol{B}$.**

A. **Prove $\boldsymbol{b \leq \beta a}$.**

Since $\beta a \leq b + c$ (applying (A.1)) , we have $b \leq \beta a$.

B. **Prove $\boldsymbol{a \leq \beta b}$.**

In the case when $\beta a < b + c + d$ we have:

$$\begin{aligned} a &< \frac{1}{\beta}(b + c + d) \quad \text{(applying(A.2), (A.5))} \\ &< \frac{1}{\beta}(b + \beta b + \frac{\beta + 1}{\beta - 1}b) \\ &= \frac{\beta + 1}{\beta - 1}b \\ &< \beta b \end{aligned}$$

ii. **Prove C is weight balanced to D.**

A. **Prove $c \leq \beta d$.**

Since $b + c \leq \beta d$ (applying (A.3)), we have $c \leq \beta d$.

B. **Prove $d \leq \beta c$.**

From (A.6), we have

$$d < \frac{\beta + 1}{\beta - 1}c < \beta c$$

iii. **Prove $A + B$ is weight balanced to $C + D$.**

A. **Prove $a + b \leq \beta(c + d)$.**

From (A.4), (A.2) and (A.7) we have:

$$\begin{aligned} d &< \frac{1}{\beta - 1}(b + c) \leq \frac{1}{\beta - 1}(\beta c + c) \\ &= \frac{\beta + 1}{\beta - 1}c < \beta c \\ &\Rightarrow \frac{1}{\beta}d < c \\ &\Rightarrow (1 + \frac{1}{\beta})d < (1 + \beta)c \\ &\Rightarrow (1 + \frac{1}{\beta} + \beta)d < \beta(c + d) + c \ \ \text{(applying (A.3))} \\ &\Rightarrow a + b + c < (1 + \frac{1}{\beta} + \beta)d < \beta(c + d) + c \\ &\Rightarrow a + b < \beta(c + d) \end{aligned}$$

B. **Prove $c + d \leq \beta(a + b)$.**

When $\beta > 2$, we have $\frac{\beta}{\beta-1} < \beta$. Thus applying (A.4) and (A.1) we have:

$$d < \frac{1}{\beta - 1}(b + c) \leq \frac{\beta}{\beta - 1}a < \beta a$$

Also we have $c \leq \beta b$ (applying (A.2)). Thus $c + d < \beta(a + b)$.

2. ***B* + *C* is too light that cannot be balanced with *D***, i.e.,

$$\beta(b + c) < d$$

In this case, we have $a < \beta(b + c) < d$ (applying (A.1) and (A.8)), which means that $a + b + c < d + \frac{1}{\beta}d < \beta d$ when $\beta > \frac{1+\sqrt{5}}{2} \approx 1.618$. This contradicts with the condition that $A + B + C$ is too heavy to D ($a + b + c > \beta d$). Thus this case is impossible.

3. ***B* + *C* is too heavy that cannot be balanced with *D***, i.e.,

$$b + c > \beta d \tag{A.8}$$

$$\Rightarrow a > \frac{1}{\beta}(b + c) > d \tag{A.9}$$

In this case, we apply the double rotation.

We need to prove the following balance conditions:

(a) **Prove *A* is weight balanced to *B*.**

i. **Prove $b < \beta a$.**

Since $\beta a > b + c$ (applying (A.1)) , we have $b < \beta a$.

ii. **Prove $a < \beta b$.**

Suppose $c = kb$, where $\frac{1}{\beta} < k < \beta$. Since $b + c > \beta d$, we have:

$$d < \frac{1 + k}{\beta}b \tag{A.10}$$

From the above inequalities, we have:

$$\begin{aligned}
a + b + c &= a + b + kb \qquad \text{(applying(A.3))}\\
&< (1 + \beta + \frac{1}{\beta})d \quad \text{(applying(A.10))}\\
&< (1 + \beta + \frac{1}{\beta}) \times \frac{1 + k}{\beta} b\\
\Rightarrow a &< \left(\frac{1 + \beta + \frac{1}{\beta}}{\beta} - 1 \right)(1 + k)b\\
&= \frac{\beta + 1}{\beta^2}(1 + k)b\\
&< \frac{(\beta + 1)^2}{\beta^2} b
\end{aligned}$$

When $\beta > \frac{7}{9}\times(\frac{\sqrt{837}+47}{54})^{-1/3}+(\frac{\sqrt{837}+47}{54})^{1/3}+\frac{1}{3} \approx 2.1479$, we have $\frac{(\beta+1)^2}{\beta} < \beta$. Hence $a < \beta b$.

(b) **Prove C is weight balanced to D.**

i. **Prove $d \leq \beta c$.**

When $\beta > \frac{1+\sqrt{5}}{2} \approx 1.618$, we have $\beta > 1 + \frac{1}{\beta}$. Assume to the contrary $c < \frac{1}{\beta}d$, we have $b < \beta c < d$. Thus:

$$b + c < (1 + \frac{1}{\beta})d < \beta d$$

, which contradicts with (A.8) that $B + C$ is too heavy to be balanced with D.

ii. **Prove $c < \beta d$.**

Plug (A.9) in (A.3) and get $b + c < (\beta + \frac{1}{\beta})d$. Recall that $\beta > 1$, we have:

$$\begin{aligned}
&\frac{1}{\beta}c + c < b + c < (\beta + \frac{1}{\beta})d\\
\Rightarrow &c < \frac{\beta^2 + 1}{\beta + 1}d < \beta d
\end{aligned}$$

(c) **Prove $A + B$ is weight balanced with $C + D$.**

i. **Prove $c + d \leq \beta(a + b)$.**

From (A.2) we have $c < \beta b$, also $d < a < \beta a$ (applying (A.9)), thus $c + d < \beta(a + b)$.

ii. **Prove $a + b \leq \beta(c + d)$.**

Recall when $\beta > \frac{1+\sqrt{5}}{2} \approx 1.618$, we have $\beta > 1 + \frac{1}{\beta}$. Applying (A.8) and (A.2) we have:

$$
\begin{aligned}
& d \leq \frac{1}{\beta}(b + c) \leq c + \frac{1}{\beta}c < \beta c \\
\Rightarrow & \frac{1}{\beta}d < c \\
\Rightarrow & (1 + \frac{1}{\beta})d < (1 + \beta)c \\
\Rightarrow & (1 + \frac{1}{\beta} + \beta)d < \beta(c + d) + c \quad \text{(applying (A.3))} \\
\Rightarrow & a + b + c < (1 + \frac{1}{\beta} + \beta)d < \beta(c + d) + c \\
\Rightarrow & a + b < \beta(c + d)
\end{aligned}
$$

Taking all the three conclusions into consideration, after either a single rotation or a double rotation, the new subtree will be rebalanced.

Then by induction we can prove Lemma 3.2.6. □

Appendix B

Implementations

B.1 The Implementation of Sweepline Paradigm

We implemented the sweepline paradigm introduced in Section 9.2.2. It only requires setting the list of points (in processing order) p, the number of points n, the initial prefix structure t_0 Init, the combine function (f) f, the fold function (ϕ) phi and the update function (h) h.

```
template⟨class Tp, class P, class T, class F,
    class Phi, class H⟩
T* sweep(P* p, size_t n, T Init, Phi phi,
         F f, H h, size_t num_blocks) {
  size_t each = ((n-1)/n_blocks);
  Tp* Sums = new Tp[n_blocks];
  T* R = new T[n+1];
  // generate partial sums for each block
  parallel_for (size_t i = 0; i ⟨ n_blocks-1; ++i) {
    size_t l = i * block_size, r = l + each;
    Sums[i] = phi(p + l, p + r);}
  // Compute the prefix sums across blocks
  R[0] = Init;
  for (size_t i = 1; i ⟨ n_blocks; ++i) {
    R[i*block_size] = f(R[(i-1)*each],
        std::move(Sums[i-1])); }
  delete[] Sums;
  // Fill in final results within each block
  parallel_for (size_t i = 0; i ⟨ n_blocks; ++i) {
    size_t l = i * each;
    size_t r = (i == n_blocks - 1) ?
    (n+1) : l + each;
  for (size_t j = l+1; j ⟨ r; ++j)
   R[j] = h(R[j-1], p[j-1]);
 }
 return R;
}
```

B.2 Using PAM for Geometry Algorithms

We give some examples of our implementations using the PAM library and sweepline paradigm. We give construction code for *RangeTree* and *RangeSwp*, as well as the query code for *RangeSwp*. They have the same inner tree structure. Note that the code shown here is almost *all* code we need to implement these data structures. Comparing with all existing libraries our implementations are much simpler and as shown in the thesis, are very efficient.

Range Tree.

```
template⟨typename X,typename Y⟩
struct RangeQuery {
 using P = pair⟨X,Y⟩;

 struct inner_map_t {
  using K = Y;
  using V = X;
  static bool comp(K a, K b) { return a ⟨ b;}
  using A = int;
  static A base(key_t k, val_t v) {return 1; }
  static A combine(A a, A b) { return a+b; }
  static A I() { return 0;} };
 using inner_map = aug_map⟨inner_map_t⟩;

 struct outer_map_t {
  using K = X;
  using V = Y;
  static bool comp(K a, K b) { return a ⟨ b;}
  using A = inner_map;
  static A base(K k, V v) {
   return A(make_pair(k.second, k.first)); }
  static A combine(A a, A b) {
   return A::union(a, b); }
  static A I() { return A();} };
 using outer_map = aug_map⟨outer_map_t⟩;
 outer_map range_tree;
```

```

  RangeQuery(vector⟨P⟩& p) {
    range_tree = outer_map(p); }
};
```

RangeSweep.

```
template⟨typename X,typename Y⟩
struct RangeQuery {
  using P = pair⟨X, Y⟩;
  using entry_t = pair⟨Y, X⟩;
  struct inner_map_t {
    using K = Y;
    using V = X;
    static bool comp(K a, K b) { return a ⟨ b;}
    using A = int;
    static A base(key_t k, val_t v) {return 1; }
    static A combine(A a, A b) { return a+b; }
    static A I() { return 0;} };
  using inner_map = aug_map⟨inner_map_t⟩;
  inner_map* ts;
  X* xs;
  size_t n;

  RangeQuery(vector⟨P⟩& p) {
    n = p.size();
    Point* A = p.data();
    auto less = [] (P a, P b)
      {return a.first ⟨ b.first;};
    parallel_sort(A, n, less);

    xs = new X[n];
    entry_t *vs = new entry_t[n];
    parallel_for (size_t i = 0; i ⟨ n; ++i) {
      xs[i] = A[i].first;
      vs[i] = entry_t(A[i].second, A[i].first); }

    auto insert = [&] (inner_map m, entry_t a) {
      return inner_map::insert(m, a);   };
    auto fold = [&] (entry_t* s, entry_t* e) {
      return inner_map(s,e);  };
    auto combine = [&] (inner_map m1, c_map m2) {
```

```
    return inner_map::union(m1, std::move(m2));};
  ts = sweep⟨inner_map⟩(vs, n, inner_map(),
                      insert, fold, combine); }

int query(X x1, Y y1, X x2, Y y2) {
  size_t l = binary_search(xs, x1);
  size_t r = binary_search(xs, x2);
  size_t left = (l⟨0) ? 0 : ts[l].aug_range(y1,y2);
  size_t right = (r⟨0) ? 0 : ts[r].aug_range(y1,y2);
  return right-left; }
```

Bibliography

[1] FaunaDB. https://fauna.com/. 1.5, 10.1, 11.4.1

[2] The problem based benchmark suite (PBBS) library. https://github.com/cmuparlay/pbbslib. II, 11.1

[3] Index spooling in microsoft sql server. https://sqlserverfast.com/epr/index-spool/. 11.2.2

[4] Lightning memory-mapped database manager (LMDB). http://www.lmdb.tech/doc/, 2015. 5.1, 11.1, 15.5, 1, 4

[5] Boost C++ Libraries. `http://www.boost.org/`, 2015. 9.1, 9.6, 15.3

[6] TPC benckmark™C standard specification revision 5.11.0. 2018. URL `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`. 1.5

[7] TPC benckmark™H standard specification revision 2.18.0. 2018. URL `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`. 1.5, 11.6.1

[8] The PAM library, an HTAP database system on TPC workloads, 2019. URL `https://github.com/cmuparlay/PAM/tree/master/tpch`. 3

[9] Umut A Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic non-zero indicators for nested parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 75–88. ACM, 2017. 5.2

[10] Stephen Adams. Implementing sets effciently in a functional language. Technical Report CSTR 92-10, University of Southampton, 1992. 1.1, 3.3.2, 15.1

[11] Stephen Adams. Efficient sets—a balancing act. *Journal of functional programming*, 3(04), 1993. 1.1, 3.3.2, 15.1

[12] Georgy Adelson-Velsky and E. M. Landis. An algorithm for the organization of information. *USSR Academy of Sciences*, 145:263–266, 1962. In Russian, English translation by Myron J. Ricci in Soviet Doklady, 3:1259-1263, 1962. 1, 2.2

[13] M AdelsonVelskii and Evgenii Mikhailovich Landis. An algorithm for the organization of information. Technical report, DTIC Document, 1963. 14.2.1

[14] Peyman Afshani, Mark De Berg, Henri Casanova, Benjamin Karsin, Colin Lambrechts, Nodari Sitchinava, and Constantinos Tsirogiannis. An efficient algorithm for the 1d total visibility-index problem. In *Proc. Algorithm Engineering and Experiments (ALENEX)*, 2017. 15.3

[15] 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. 15.3, 15.4

[16] Alok Aggarwal and Jeffrey S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1988. 14, 14.1.1

[17] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Ó'Dúnlaing, and Chee Yap. Parallel computational geometry. *Algorithmica*, 3(1-4):293–327, 1988. 1.5, 9.1, 9.4.1, 15.3

[18] Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 84–95, 2014. 1.5, 10.1, 11.4.1

[19] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004. 15.5

[20] Thomas D Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proc. the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017. 15.3

[21] Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. Geometric algorithms for private-cache chip multiprocessors. In *European Symposium on Algorithms*, pages 75–86. Springer, 2010. 15.3

[22] Yaroslav Akhremtsev and Peter Sanders. Fast parallel operations on search trees. In *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*, pages 291–300, 2016. 15.2, 15.4

[23] 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. ACM, 2016. ISBN 978-1-4503-3716-8. 15.4

[24] Mahdi Amani, Kevin A. Lai, and Robert E. Tarjan. Amortized rotation cost in AVL trees. *Information Processing Letters*, 116(5):327–330, 2016. 14.2.1

[25] Flurry analytics, -. URL `https://developer.yahoo.com/flurry/docs/analytics/`. 1.2, 6.1.3, 15.4

[26] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media, Inc., 2010. 5.1, 15.5

[27] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 32(6), 2003. 15.3

[28] Lars Arge and Norbert Zeh. Simple and semi-dynamic structures for cache-oblivious planar orthogonal range searching. In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 158–166. ACM, 2006. 15.3

[29] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 346–357, 1999. 15.3

[30] Lars Arge, Klaus H Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002. 10.1, 11.4.1

[31] Lars Arge, Gerth Stølting Brodal, Rolf Fagerberg, and Morten Laustsen. Cache-oblivious planar orthogonal range searching and counting. In *Proceedings of the twenty-first annual symposium on Computational geometry*, pages 160–169. ACM, 2005. 15.3

[32] Mikhail J. Atallah and Michael T. Goodrich. Efficient plane sweeping in parallel. In *Proceedings of the Second Annual ACM SIGACT/SIGGRAPH Symposium on Computational Geometry, Yorktown Heights, NY, USA, June 2-4, 1986*, pages 216–225, 1986. 1.5, 9.1, 9.4.1, 15.3

[33] Mikhail J Atallah and Michael T Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 3(4):492–507, 1986. 15.3

[34] Mikhail J Atallah, Michael T Goodrich, and S Rao Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. In *Aegean Workshop on Computing*, 1988. 15.3

[35] Mikhail J. Atallah, Richard Cole, and Michael T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. 18(3):499–532, June 1989. 1.5, 9.1, 15.3

[36] Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A Ross. Path processing using solid state storage. In *Proceedings of the 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2012)*, number CONF, 2012. 15.5

[37] Kamil Bajda-Pawlikowski, Daniel J Abadi, Avi Silberschatz, and Erik Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1165–1176. ACM, 2011. 15.5

[38] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. Faster adaptive set intersections for text searching. In *International Workshop on Experimental and Efficient Algorithms*, pages 146–157. Springer, 2006. 13.1.1

[39] 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. 15.4

[40] 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 *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–369, 2017. 15.4

[41] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4), 1972. 14.2.1

[42] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972. 1, 2.2

[43] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4): 264–275, 1996. 1.3, 5.1, 15.5

[44] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, 1990. 15.3

[45] Amir M. Ben-Amram. What is a “Pointer Machine”? *SIGACT News*, 26(2):88–95, June 1995. ISSN 0163-5700. doi: 10.1145/202840.202846. URL `http://doi.acm.org/10.1145/202840.202846`. 2.3, 5.1

[46] 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 *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 145–156, 2016. 14

[47] Naama Ben-David, David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. k-abortable objects: progress under high contention. In *International Symposium on Distributed Computing*, pages 298–312. Springer, 2016. 12.3

[48] Naama Ben-David, Guy Blelloch, Yihan Sun, and Yuanhao Wei. Efficient single writer concurrency. In *arXiv preprint arXiv:1803.08617*, 2018. 1.6, 10.3.1, 12.3

[49] Naama Ben-David, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2018. 14

[50] Naama Ben-David, Guy Blelloch, Yihan Sun, and Yuanhao Wei. Multiversion concurrency with bounded delay and precise garbage collection. In *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2019. 1, 1.5, 1.6, 10.3.1, 12.1, 12.3, 12.3, 12.5, 16

[51] Samuel W Bent, Daniel D Sleator, and Robert E Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985. 15.1

[52] 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, 1.5, 9.1

[53] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975. 1.7

[54] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975. 15.3

[55] Jon Louis Bentley. Algorithms for klee's rectangle problems. Technical report, Technical Report, Computer, 1977. 1.7

[56] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5), 1979. 1, 1.5, 9.1, 15.3

[57] Jon Louis Bentley and Jerome H Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409, 1979. 1.5, 9.1, 15.3

[58] Jon Louis Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, (7):571–577, 1980. 1.5, 9.1, 15.3

[59] 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. 1.5, 2.5

[60] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981. ISSN 0360-0300. doi: 10.1145/356842.356846. URL `http://doi.acm.org/10.1145/356842.356846`. 1.3, 11.1

[61] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. ISSN 0362-5915. doi: 10.1145/319996.319998. URL `http://doi.acm.org/10.1145/319996.319998`. 1.3, 1.5, 5.1, 11.1, 12.1

[62] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. Concurrency control and recovery in database systems. 1987. 1.5, 11.1, 15.5

[63] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-a transactional record manager for shared flash. In *Innovative Data Systems Research (CIDR)*, volume 11,

pages 9–20, 2011. 11.1, 15.5

[64] Philip A Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1295–1309. ACM, 2015. 5.1, 11.1, 15.5

[65] Elisa Bertino and Won Kim. Indexing techniques for queries on nested objects. *IEEE Transactions on knowledge and data engineering*, 1(2):196–214, 1989. 15.5

[66] Juan Besa and Yadran Eterovic. A concurrent red-black tree. *J. Parallel Distrib. Comput.*, 73(4):434–449, 2013. 15.4

[67] Philip Bille, Inge Li Gørtz, and Søren Vind. Compressed data structures for range reporting. In *Proc. International Conference on Language and Automata Theory and Applications (LATA)*, 2015. 15.3

[68] Gabriele Blankenagel and Ralf Hartmut Güting. External segment trees. *Algorithmica*, 12(6):498–532, 1994. 15.3

[69] Guy Blelloch, Daniel Ferizovic, and Yihan Sun. Parallel ordered sets using join. *arXiv preprint:1602.02120*, 2016. 1.6

[70] Guy E. Blelloch and Yan Gu. Improved parallel cache-oblivious algorithms for dynamic programming and linear algebra. *arXiv preprint arXiv:1809.09330*, 2018. 14

[71] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 16–26, 1998. 1.1, 3, 3.3.2, 15.2, 15.4

[72] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, volume 47, pages 181–192, 2012. II, 6.2

[73] 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. 14, 14.1.2

[74] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM, 2016. 1.6, 3.3.4, 9.2.3, 11.1, 11.4, 14.3, 15.4

[75] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Yan Gu, and Julian Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, pages 14:1–14:18, 2016. 14, 14.1.1

[76] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478, 2016. 1.6

[77] 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. 1.6

[78] Guy E. Blelloch, Yan Gu, and Yihan Sun. Efficient construction of probabilistic tree embeddings. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2017. 1.6

[79] Guy E Blelloch, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The parallel persistent memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 14

[80] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2018. 1.6, 14, 15.3

[81] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. Optimal parallel algorithms in the binary-forking model. *CoRR*, abs/1903.04650, 2019. URL `http://arxiv.org/abs/1903.04650`. 2.1, 3.3.2, 4.3, 16

[82] Norbert Blum and Kurt Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11(3):303–320, 1980. 2.2, 3.2.3

[83] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 27(1):202–229, 1998. 2.1

[84] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. 11.1

[85] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005. 11.1

[86] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Usenix Conference on File and Storage Technologies*, volume 215, 2003. 15.5

[87] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013. 15.5

[88] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974. 2.1

[89] Nieves R Brisaboa, Guillermo De Bernardo, Roberto Konow, Gonzalo Navarro, and Diego Seco. Aggregated 2d range queries on clustered points. *Information Systems*, 2016. 15.3

[90] 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. 15.2, 15.4

[91] Mark R Brown and Robert E Tarjan. A fast merging algorithm. *Journal of the ACM (JACM)*, 26(2):211–226, 1979. 1.1, 3.3.2, 15.2

[92] Trevor Brown. Lock-free chromatic trees in c++. `https://bitbucket.org/trbot86/implementations/src/`, 2016. 10.3.2

[93] Trevor Brown. Lock-free chromatic trees in c++. `https://bitbucket.org/trbot86/implementations/src/`, 2016. 10.3.2

[94] 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. 15.4

[95] 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. 15.4

[96] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2014. 10.3.2

[97] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 729–738, 2008. 10.1, 11.4.1

[98] CGAL. CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`. 9.1, 9.6, 15.3

[99] 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. 15.4

[100] Chee Yong Chan and Yannis E Ioannidis. Hierarchical cubes for range-sum queries. In *Proceedings of the 25th VLDB Conference*, pages 675–686, 1999. 1.5, 9.1

[101] Bernard Chaselle. Intersecting is easier than sorting. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 125–134, New York, NY, USA, 1984. ACM. ISBN 0-89791-133-4. doi: 10.1145/800057.808674. URL `http://doi.acm.org/10.1145/800057.808674`. 1.5, 9.1, 9.4.1, 15.3

[102] Bernard Chazelle and Herbert Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM (JACM)*, 39(1):1–54, 1992. 15.3

[103] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1986. 9.1, 15.3

[104] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. The MemSQL query optimizer: A modern optimizer for real-time analytics in a distributed database. *Proceedings of the VLDB Endowment*, 9(13): 1401–1412, 2016. 11.6.2

[105] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Proc. Conference on Innovative Data Systems Research (CIDR)*, 2011. 14.1.2

[106] Siu Wing Cheng and Ravi Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters*, 1990. 1.5, 9.1, 15.3

[107] Yi-Jen Chiang. Experiments on the practical i/o efficiency of geometric algorithms: Distribution sweep versus plane sweep. *Computational Geometry*, 1998. 9.2.3, 15.3

[108] Marek Chrobak, Tomasz Szymacha, and Adam Krawczyk. A data structure useful for finding hamiltonian cycles. *Theoretical Computer Science*, 71(3):419–424, 1990. 15.1

[109] Sailesh Chutani, Owen T Anderson, Michael L Kazar, Bruce W Leverett, W Anthony Mason, Robert N Sidebotham, et al. The episode file system. In *USENIX Winter 1992 Technical Conference*, pages 43–60, 1992. 15.5

[110] Rosetta Code. Inverted index. URL `https://rosettacode.org/wiki/Inverted_index`. 13.2.1

[111] Nachshon Cohen and Erez Petrank. Efficient memory management for lock-free data structures with optimistic access. pages 254–263. ACM, 2015. 1.5, 12.1

[112] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload CH-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011. 11.6.1

[113] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. The mixed workload ch-benchmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 8. ACM, 2011. 5

[114] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3 (12):655–657, December 1960. 5, 5.2

[115] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009. 14.1.2

[116] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *Acm Sigmod Record*, volume 14, pages 268–279. ACM, 1985. 11.1

[117] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (second edition).* MIT Press and McGraw-Hill, 2001. 4.1, 4.1, 4.4

[118] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009. 1.2, 1.7, 8, 8.1, 9.5, 15.3

[119] Costin S. C# based interval tree. `https://code.google.com/archive/p/intervaltree`, 2012. 8

[120] AN Craig, GR Soules, JD Goodson, and GR Strunk. Metadata efficiency in versioning file systems. In *USENIX Conference on File and Storage Technologies*, 2003. 15.5

[121] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A Wood. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984. 10.1, 11.4.1

[122] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 918–934, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314598. URL `http://doi.acm.org/10.1145/3314221.3314598`. 11.6.4, 16

[123] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1243–1254, 2013. 5, 15.5

[124] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013. 11.6.1

[125] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *Proc. of the VLDB Endowment*, 4(4):255–266, 2011. 15.4

[126] 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. 15.4

[127] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989. 1.3, 2.3, 3, 9.3.2

[128] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. Supporting table partitioning by reference in oracle. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1111–1122. ACM, 2008. 15.5

[129] Herbert Edelsbrunner. Dynamic rectangle intersrction searching. Technical Report Institute for Technical Processing Report 47, Technical University of Graz, Austria, 1980. 8

[130] Herbert Edelsbrunner. A new approach to rectangle intersections part i. *International Journal of Computer Mathematics*, 1983. 1.5, 9.1, 15.3

[131] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 1981. 1.5, 9.1, 15.3

[132] 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. 15.4

[133] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012. 5, 15.5

[134] Preparata Franco and Michael Ian Preparata Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 1985. 4.1, 4.4, 8

[135] 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. 15.1, 15.4

[136] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. Innodb database forensics. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 1028–1036. IEEE, 2010. 15.5

[137] 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. 15.4

[138] Georges Gardarin, Eric Simon, and Lionel Verlaine. Querying real time relational data bases. In *ICC (2)*, pages 757–761, 1984. 15.5

[139] Michael T Goodrich. Intersecting line segments in parallel with an output-sensitive number of processors. *SIAM Journal on Computing*, 1991. 15.3

[140] Michael T Goodrich and Darren Strash. Priority range trees. In *International Symposium on Algorithms and Computation (ISAAC)*, 2010. 15.3

[141] Michael T. Goodrich, Mujtaba R. Ghouse, and J Bright. Sweep methods for parallel computational geometry. *Algorithmica*, 15(2):126–153, 1996. 1.5, 9.1, 15.3

[142] Goetz Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4): 203–402, 2011. 11.3

[143] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969. 2.1

[144] 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. 15.4

[145] Yan Gu. *Write-Efficient Algorithms*. PhD Thesis, Carnegie Mellon University, 2018. 14

[146] Yan Gu, Julian Shun, Yihan Sun, and Guy E Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015. 1.6

[147] Yan Gu, Yihan Sun, and Guy E Blelloch. Algorithmic building blocks for asymmetric memories. In *European Symposium on Algorithms (ESA)*, 2018. 14.1.2

[148] Yan Gu, Yihan Sun, and Guy E Blelloch. Algorithmic building blocks for asymmetric memories. In *26th Annual European Symposium on Algorithms (ESA 2018)*, 2018. 1.6

[149] Yan Gu, Yihan Sun, and Guy E Blelloch. Algorithmic building blocks for asymmetric memories. *arXiv preprint arXiv:1806.10370*, 2018. 1.6

[150] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE, 1978. 3.4

[151] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. 1978. 14.2.1

[152] 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. 15.4

[153] Bernhard Haeupler, Siddhartha Sen, and Robert E Tarjan. Rank-balanced trees. In *Workshop on Algorithms and Data Structures*, pages 351–362. Springer, 2009. 3.4

[154] Chaim-Leib Halbet and Konstantin Tretyakov. Python based interval tree. `https://github.com/chaimleib/intervaltree`, 2015. 8.1, 8.1.1

[155] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010. 1.5, 10.1, 11.4.1

[156] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. 12.2

[157] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914. 12.2

[158] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*

(TOPLAS), 12(3):463–492, 1990. 12.2

[159] Yoichi Hirai and Kazuhiko Yamamoto. Balancing weight-balanced trees. *Journal of Functional Programming*, 21(03):287–307, 2011. 5, 15.1

[160] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994. 15.5

[161] 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. ACM, 1997. ISBN 0-89791-911-4. 1.5, 9.1, 15.4

[162] Jeffrey A Hoffer, Ramesh Venkataraman, and Heikki Topi. *Modern Database Management, 11/E.* Prentice Hall, 2009. 15.5

[163] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proc. of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 351–363, 1986. 5, 6.2

[164] 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. 3, 3.3.2, 15.2

[165] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proc. of the VLDB Endowment*, 8(3):293–304, 2014. 13.1.1, 15.4

[166] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org. 7.1

[167] Rico Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '17, 2017. 15.3

[168] Riko Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 247–254, 2017. 14

[169] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992. 2.1

[170] Elizabeth Johnson and Dennis Gannon. HPC++: experiments with the parallel standard template library. In *International Conference on Supercomputing (ICS)*, pages 124–131, 1997. 15.2

[171] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795, 9781420082791. 5, 5.2

[172] Ibrahim Kamel and Christos Faloutsos. *Parallel R-trees.* 1992. 1.5, 9.1, 15.3

[173] Haim Kaplan and Robert Endre Tarjan. Purely functional representations of catenable sorted lists. In *Proc. ACM Symposium on the Theory of Computing (STOC)*,

pages 202–211, 1996. 5.1

[174] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: execute-verify replication for multi-core servers. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 237–250, 2012. 15.5

[175] J Katajainen. Efficient parallel algorithms for manipulating sorted sets. In *Proc. of Computer Science Conference*. University of Canterbury, 1994. 15.2

[176] Jyrki Katajainen, Christos Levcopoulos, and Ola Petersson. *Space-efficient parallel merging*. Springer, 1992. 15.2

[177] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-8959-6. doi: 10.1109/ICDE.2011.5767867. URL http://dx.doi.org/10.1109/ICDE.2011.5767867. 5

[178] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *IEEE International Conference on Data Engineering (ICDE)*, pages 195–206, 2011. 5, 11.6, 11.6.3

[179] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. Parallel multi-dimensional range query processing with r-trees on gpu. *Journal of Parallel and Distributed Computing*, 73(8):1195 – 1207, 2013. ISSN 0743-7315. 15.4

[180] 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. 8, 15.4

[181] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version stm algorithm. In *Proc. International Conference on Distributed Computing and Networking (ICDN)*, pages 212–226, 2014. ISBN 978-3-642-45248-2. 1.3, 1.5, 5.1, 11.1, 12.1

[182] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980. 15.2, 15.4

[183] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013. 1, 11.1

[184] Kim S. Larsen. AVL trees with relaxed balance. *J. Comput. Syst. Sci.*, 61(3):508–522, 2000. 15.2, 15.4

[185] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M Patel, and Mike Zwilling. High-performance concurrency control mechanisms for

main-memory databases. *VLDB Endowment*, 5(4):298–309, 2011. 5, 15.5

[186] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37. ACM, 2009. 14.1.2

[187] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W Keller. Energy management for commercial servers. *Computer*, 36(12): 39–48, 2003. 1

[188] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, 2014. 11.6.2

[189] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of practical synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016*, pages 3:1–3:8, 2016. 10.3.2

[190] 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. 10.3.2, 15.4

[191] LevelDB, -. URL `leveldb.org`. 1.2, 6.1.3, 15.4

[192] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems*, page 4. ACM, 2014. 1

[193] Yinan Li and Jignesh M Patel. Widetable: An accelerator for analytical data processing. *Proceedings of the VLDB Endowment*, 7(10):907–918, 2014. 11.1, 15.5

[194] Raymond A Lorie. Physical integrity in a large segmented database. *ACM Transactions on Database Systems (TODS)*, 2(1):91–104, 1977. 15.5

[195] George S Lueker. A data structure for orthogonal range queries. In *Symp. Foundations of Computer Science*, 1978. 3.4, 9.3.1

[196] David Maier, David Maier, and Jacob Stein. Indexing in an object-oriented dbms. In *Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 171–182. IEEE Computer Society Press, 1986. 15.5

[197] Krishna T Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin C Lee, and Mark Horowitz. Rethinking DRAM power modes for energy proportionality. In *IEEE/ACM International Symposium on Microarchitecture*, pages 131–142, 2012. 1

[198] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *ACM European Conference on Computer Systems*, pages 183–196, 2012. 10.3.2

[199] Mark De Berg Mark, Mark Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. *Computational Geometry.* Springer, 2000. 8, 15.3, 15.4

[200] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010. 3.3.2, 15.1

[201] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. In *Proc. Symposium on Operating Systems Principles (SOSP)*, 2015. 12.1

[202] Edward M McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical report, 1980. 15.3

[203] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985. 1.2

[204] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 1985. 15.3

[205] Mark McKenney and Tynan McGuire. A parallel plane sweep algorithm for multi-core systems. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*, pages 392–395. ACM, 2009. 1.5, 9.1, 9.2.3, 15.3

[206] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, October 1998. 1.5, 12.1

[207] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *Ottawa Linux Symposium*, July 2001. 1.5, 12.1

[208] Kurt Mehlhorn and Stefan Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. 1994. 15.3

[209] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-56329-1. 15.1

[210] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel & Distributed Systems*, (6):491–504, 2004. 1.5, 12.1

[211] C Mohan, Hamid Pirahesh, and Raymond Lorie. *Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions*, volume 21. 1992. 1.5, 11.1, 15.5

[212] David R Musser, Gillmer J Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library.* Addison-Wesley Professional, 2009. 7.1

[213] 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. 15.2, 15.4

[214] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015. 1, 1.3, 1.5, 5.1, 11.1, 11.1, 12.1, 15.5

[215] 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`. 9.6, 15.4

[216] Jürg Nievergelt and Edward M Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1), 1973. 14.2.1

[217] Jürg Nievergelt and Edward M. Reingold. Binary search trees of bounded balance. *SIAM J. Comput.*, 2(1):33–43, 1973. 1, 2.2, 3.2.3

[218] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees. *Acta informatica*, 33(6):547–557, 1996. 10.3.2

[219] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6. 2.3, 5.1

[220] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. 15.5

[221] Oracle. Oracle nosql, -. URL `https://www.oracle.com/database/nosql/index.html`. 1.2, 6.1.3, 15.4

[222] Mark H Overmars. Geometric data structures for computer graphics: an overview. In *Theoretical foundations of computer graphics and CAD*, pages 21–49. Springer, 1988. 15.3

[223] M Tamer Özsu. A survey of rdf data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016. 15.5

[224] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, 1999. 13.1.1

[225] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 214–221. ACM, 1993. 15.3

[226] Christos H Papadimitriou and Paris C Kanellakis. On concurrency control by multiple versions. *ACM Transactions on Database Systems (TODS)*, 1984. 1.3, 1.5, 5.1, 11.1, 12.1

[227] Heejin Park and Kunsoo Park. Parallel algorithms for red-black trees. *Theoretical Computer Science*, 262(1–2):415–435, 2001. 15.2, 15.4

[228] 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. 15.2, 15.4

[229] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 16–25. ACM, 2010. 1.3, 1.5, 12.1

[230] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. `https://www.gartner.com/doc/2657815/`, 2014. 1.5, 11.1

[231] Chuck Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008. 7.1

[232] Nicholas Pippenger. Pure versus impure lisp. *ACM Trans. Program. Lang. Syst.*, 19 (2):223–238, March 1997. 2.3, 5.1

[233] Dan RK Ports and Kevin Grittner. Serializable snapshot isolation in PostgreSQL. *VLDB Endowment*, 5(12):1850–1861, 2012. 1, 5, 11.1, 15.5

[234] 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. 15.4

[235] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3), 2009. 14.1.2

[236] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets:*. Cambridge University Press, 10 2011. 13.1.1

[237] David P. Reed. Naming and synchronization in a decentralized computer system, 1978. 1.3, 1.5, 5.1, 11.1, 12.1

[238] Colin Reid, Philip A Bernstein, Ming Wu, and Xinhao Yuan. Optimistic concurrency control by melding trees. *Proceedings of the VLDB Endowment*, 4(11), 2011. 10.1, 11.1, 11.4.1, 6

[239] RocksDB, -. URL `rockdb.org`. 1.2, 6.1.3, 15.4

[240] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *ACM Sigmod Record*, 1985. 15.3

[241] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Proc. of the Meeting on Algorithm Engineering & Expermiments (ALENEX)*, pages 71–83, 2007. 13.1.1

[242] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986. 1, 1.3, 2.3, 9.2.1

[243] Benjamin Schlegel, Tu Dresden, Thomas Willhalm, Wolfgang Lehner, and Tu Dresden. Fast sorted-set intersection using simd instructions. In *In ADMS Workshop*, 2011. 13.1.1

[244] Jens M Schmidt. Interval stabbing problems in small integer ranges. In *International Symposium on Algorithms and Computation*, 2009. 15.3

[245] Bernhard Seeger and Per-Åke Larson. Multi-disk b-trees. In *ACM SIGMOD Record*, 1991. 15.3

[246] Raimund Seidel and Celcilia R. Aragon. Randomized search trees. 16:464–497, 1996. 1, 2.2, 14.2.1, 15.1

[247] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. Palm: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors. *VLDB Endowment*, 4(11):795–806, 2011. 1.5, 10.1, 11.4.1

[248] Nikita Shamgunov. The memsql in-memory database system. In *IMDMat VLDB*, 2014. 1, 11.1, 11.1, 11.6

[249] M. I. Shamos and D. Hoey. Geometric intersection problems. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 208–215, Oct 1976. 1.3, 1.5, 2.4, 9.1, 9.2.1

[250] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, August 2013. ISSN 2150-8097. 1.2, 6.1.3

[251] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 731–742, 2012. 5, 15.5

[252] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: the multi-core standard template library. In *Proc. European Conf. on Parallel Processing (Euro-Par)*, pages 682–694, 2007. 7.1, 7.1

[253] Nodari Sitchinava. Computational geometry in the parallel external memory model. *SIGSPATIAL Special*, 4(2), 2012. 15.3

[254] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2), 1985. ISSN 0001-0782. doi: 10.1145/2786.2793. URL `http://doi.acm.org/10.1145/2786.2793`. 1, 1.1

[255] Benjamin Sowell, Wojciech Golab, and Mehul A Shah. Minuet: A scalable distributed multiversion b-tree. *VLDB Endowment*, 5(9):884–895, 2012. 1.3, 5.1, 15.5, 4

[256] Jagannathan Srinivasan, Souripriya Das, Chuck Freiwald, Eugene Inseok Chong, Mahesh Jagannath, Aravind Yalamanchi, Ramkumar Krishnan, Anh-Tuan Tran, Samuel DeFazio, and Jayanta Banerjee. Oracle8i index-organized table and its application to new domains. In *VLDB*, pages 285–296, 2000. 11.6.2

[257] Milan Straka. Adams' trees revisited. In *Trends in Functional Programming*, pages 130–145. Springer, 2012. 5, 15.1

[258] Yihan Sun and Guy E Blelloch. Parallel range and segment queries with augmented maps. *arXiv preprint:1803.08621*, 2018. 1.6

[259] Yihan Sun and Guy E Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019. 1.2, 1.6

[260] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: Parallel augmented maps. *arXiv preprint: 1612.05665*, 2016. 1.6

[261] Yihan Sun, Guy Blelloch, and Daniel Ferizovic. The PAM library. `https://github.com/cmuparlay/PAM`, 2018. 1, 6, 9.6

[262] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018. 1.6, 6, 6.1, 6.1.4, 9.1, 9.5, 9.6, 11.1

[263] Yihan Sun, Guy E Blelloch, Andrew Pavlo, and Wan Shen Lim. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *PVLDB*, 2020. 1.6

[264] Gabriel Tanase, Chidambareswaran Raman, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Associative parallel containers in STAPL. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 156–171, 2007. 15.2

[265] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983. ISBN 0-89871-187-8. 1.1, 3.2.2, 14.2.1, 15.1

[266] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012. 1.5, 10.1, 11.4.1, 6

[267] Ziqi Wang. Index microbench. `https://github.com/wangziqi2016/index-microbench`, 2017. 10.3.2

[268] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488. ACM, 2018. 10.3.2

[269] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery.* Elsevier, 2001. 1.5, 11.1, 15.5

[270] Ron Wein. Efficient implementation of red-black trees with split and catenate operations. Technical report, Tel-Aviv University, 2005. 15.1

[271] Wikimedia Foundation. Wikepedia:database download. `https://en.wikipedia.org/wiki/Wikipedia:Database_download`, 2016. 13.2.1

[272] Dan E Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 1985. 15.3

[273] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *VLDB Endowment*, 10: 781–792, March 2017. 1, 1.3, 11.1

[274] Boseon Yu, Hyunduk Kim, Wonik Choi, and Dongseop Kwon. Parallel range query processing on R-tree with graphics processing unit. In *Dependable, Autonomic and Secure Computing (DASC)*, 2011. 15.3

[275] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over dht. In *IPTPS*, 2006. 15.3

[276] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006. ISSN 0360-0300. 13.1.1