SPAA2020
Tutorial

Parallel Balanced Binary trees in Shared-Memory

Yihan Sun
University of California, Riverside

What's in this tutorial

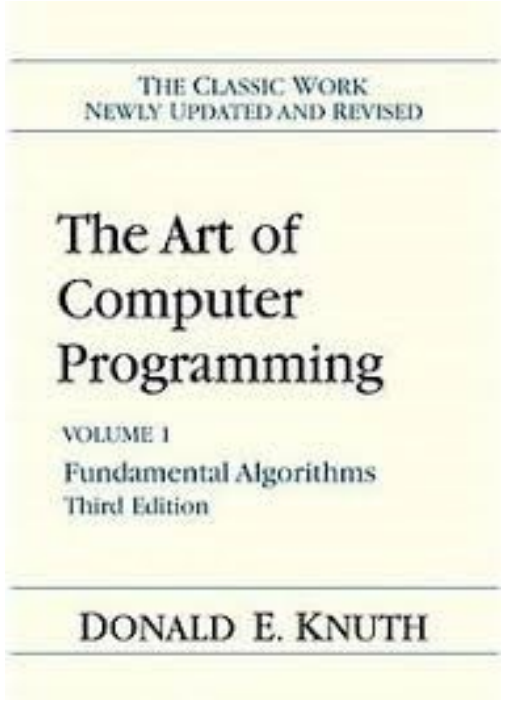
- Algorithms and implementation details about parallel balanced binary trees (P-trees)
  - Simple with various functionalities
- An open-source parallel library (PAM) and example code to use it
- Applications that can be solved using the algorithms and the library in this tutorial

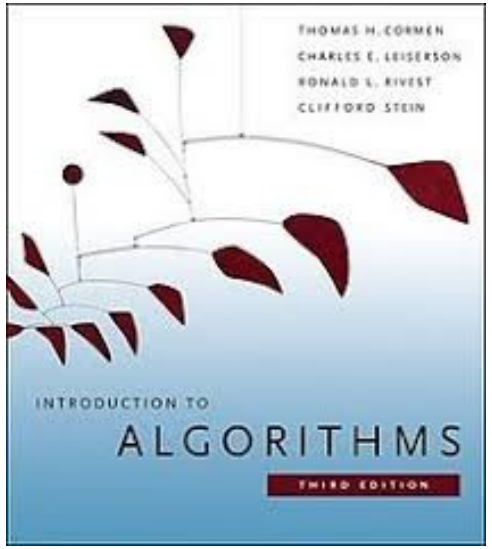
Trees

- Trees are fundamental data structures for organizing data
- Taught in entry-level undergraduate courses
- In real-world applications, things are more complicated… Especially in parallel

[Sedgewick and Wayne]
Sec. 3.2 Binary Search Trees
Sec. 3.3 Balanced Search Trees

[TAOCP]
2.3 Trees
6.2.2. Binary Tree Searching
6.2.3. Balanced Trees
6.2.4. Multiway Trees





[CLRS]
12 Binary Search Trees
13 Red-Black Trees
14.3 Interval trees

Applications Using Trees

Document Search Engine

Find information about
“balanced” and “binary”

1,234,567 results

Doc 1: Balanced binary tree

A **binary** tree is **balanced** if it keeps its height small …

Doc 2: AVL tree

AVL tree is a **balanced binary** search tree structure …

```
Struct Doc {
  int l;
  pair<Word, Weight>* w;
};
class doc_tree {
  void build(Doc* d);
  Doc_set search(Word w);
  Doc_set and_search(Word* w);
  Doc_set or_search(Word* w);
  void add_doc(Doc d);
  ……
};
```

Searching and updating may need to be done concurrently

Applications using trees

Databases

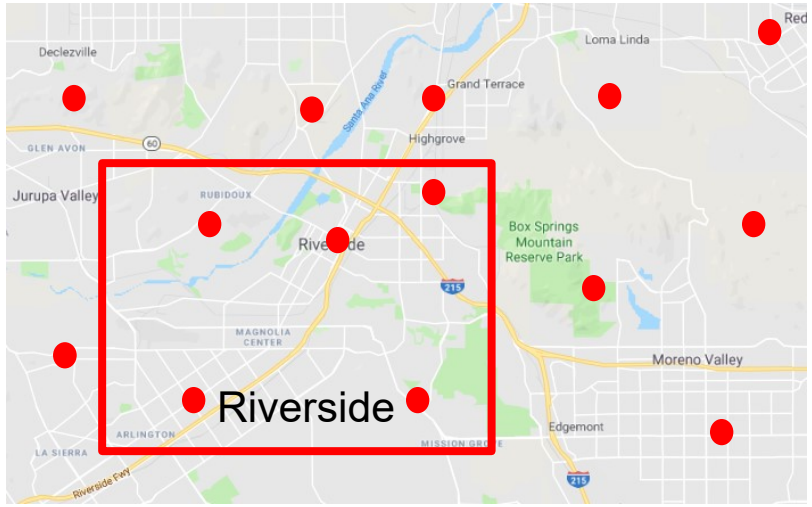
Find all young CS students with good grade

```
select
    name
from
    students
where
    age < 25
    and major = ‘CS’
    and grade >= ‘A’
```

```
Struct Student {
  id name, grade, age, major, …
};
class database {
  void build(Doc* d);
  Student* search(int id);
  Student* fitler(function f);
  void add_student(Student s);
  ……
};
```

Applications Using Trees

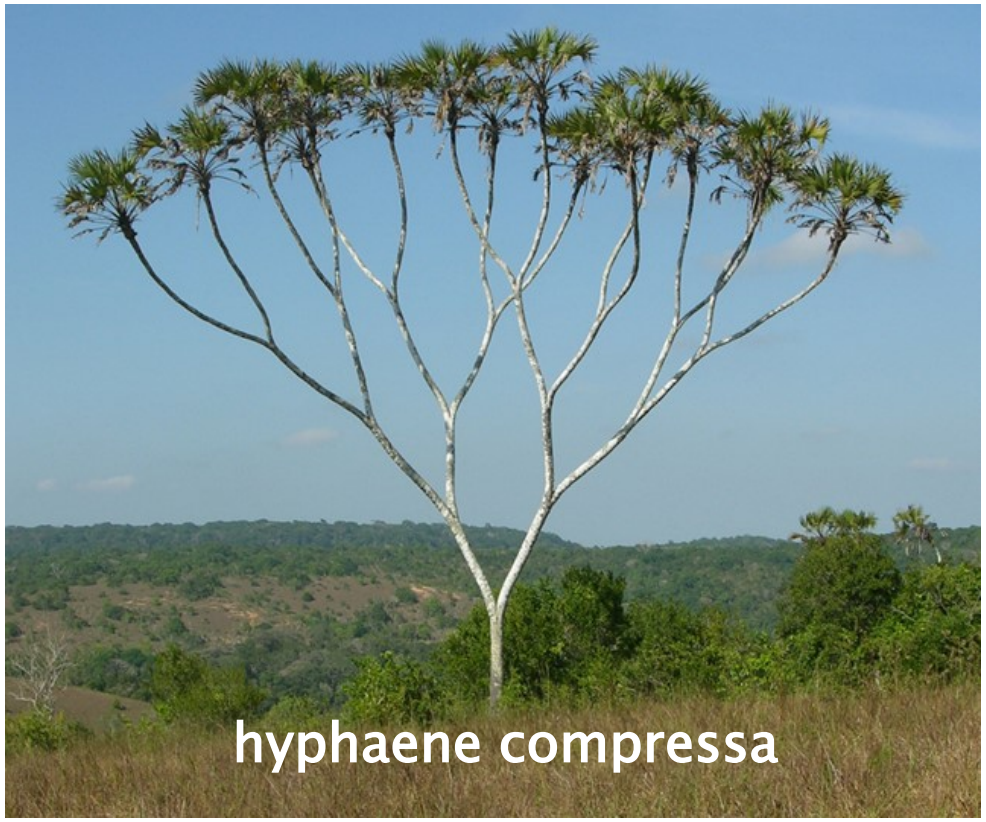
Geometric queries
(A 2D range query)
Find average temperature
in Riverside



```
struct Point {
  X x; Y y; Weight w;
};
class range_tree {
  void build(Point* p);
  Double ave_weight_search(X x1,
X x2, Y y1, Y y2);
  int count_search(X x1, X x2, Y
y1, Y y2);
  int list_all_search(X x1, X x2,
Y y1, Y y2);
  void insert(Point p);
  range_tree filter(func f);
  Point* output();
  void update(X x, Y y, Weight w);
};
```

Balanced Binary Trees

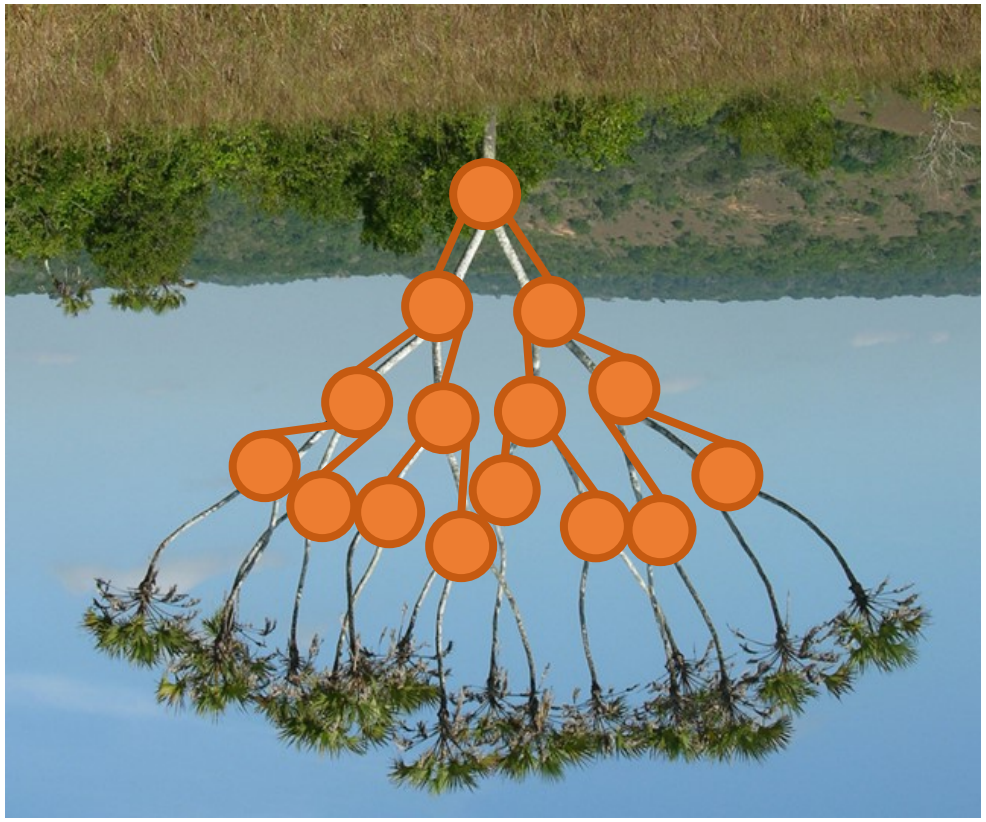
- Binary: each tree node has at most two children
- Balanced: the tree has bounded height
  - Usually $O(\log n)$ for size n



A wild balanced binary tree

Balanced Binary Trees

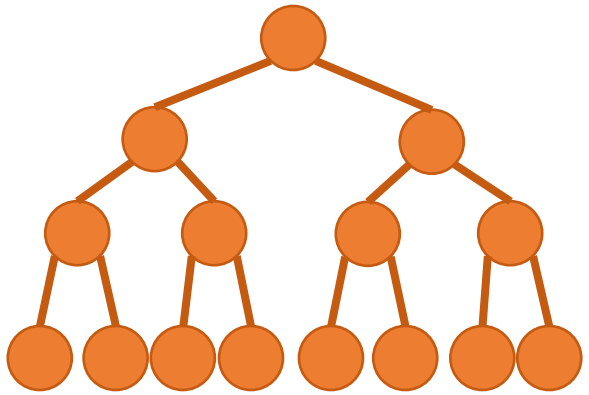
- Binary: each tree node has at most two children
- Balanced: the tree has bounded height
  - Usually $O(\log n)$ for size n



A wild balanced binary tree

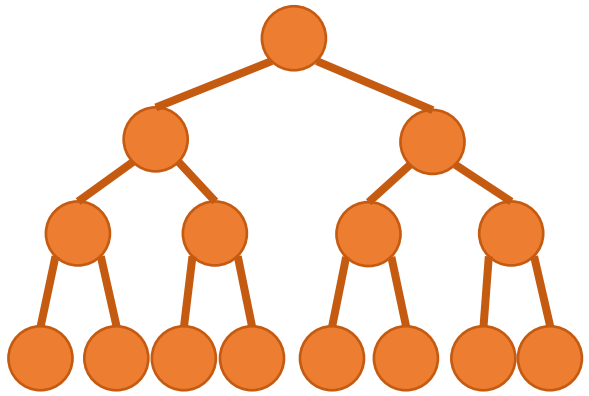
Balanced Binary Trees

- Binary: each tree node has at most two children
- Balanced: the tree has bounded height
  - Usually $O(\log n)$ for size n



An abstract balanced binary tree

Balanced Binary Trees



- Balancing schemes: invariants to keep the tree balanced and to bound the tree height
- We discuss four standard balancing schemes

Applications Using Trees

Document Search Engine

Find information about "balanced" and "binary"

Databases

Find all young CS students with good grade

Geometric queries

(A 2D range query)
Find average temperature in Riverside area: 62 F

Balanced A

1,234,567 r

Doc 1: Bala

A **binary** tre
keeps its he

Doc 2: AVL

AVL tree is a
search tree

What we want

Elegance – Framework

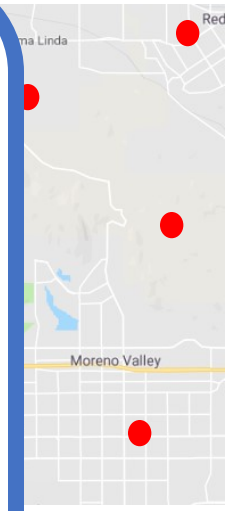
Generic for balancing schemes
Generic for applications

Massive Data - Performance

Parallelism and concurrency
Efficiency both in theory and in practice

Comprehensive Queries – Functionality

Range queries, bulk updates, …
Augmentation, …
Dynamicity, multi-versioning, …



What does P-tree look like?

Elegance

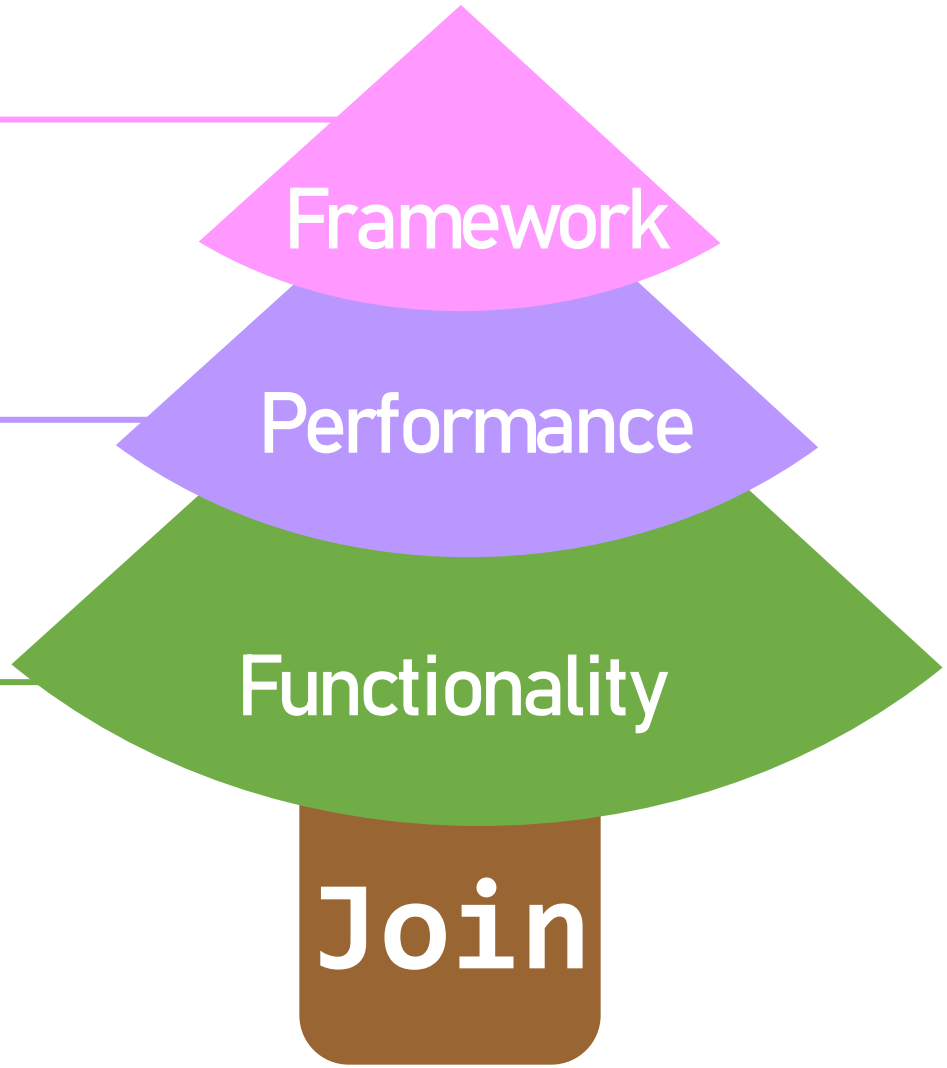
- Generic for balancing schemes
- Generic for applications

Massive Data

- Parallelism and concurrency
- Efficiency both in theory and in practice

Comprehensive Queries

- Range queries, bulk updates, …
- Augmentation, …
- Dynamicity, multi-versioning, …



(A primitive for trees)

1 Generic for balancing schemes

All algorithms except join are identical across balancing schemes

One algorithm for multiple balancing schemes!

2 Generic for applications

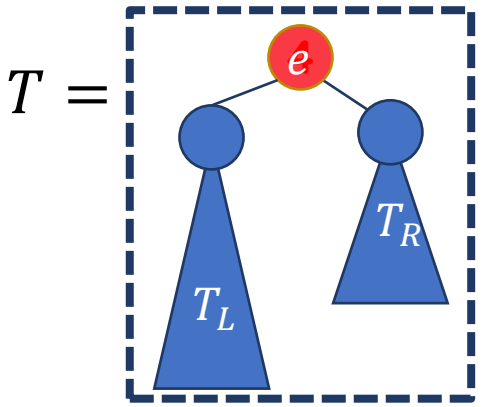
Multiple applications based on the same tree structure

One tree for different problems!

A primitive for trees: `Join`

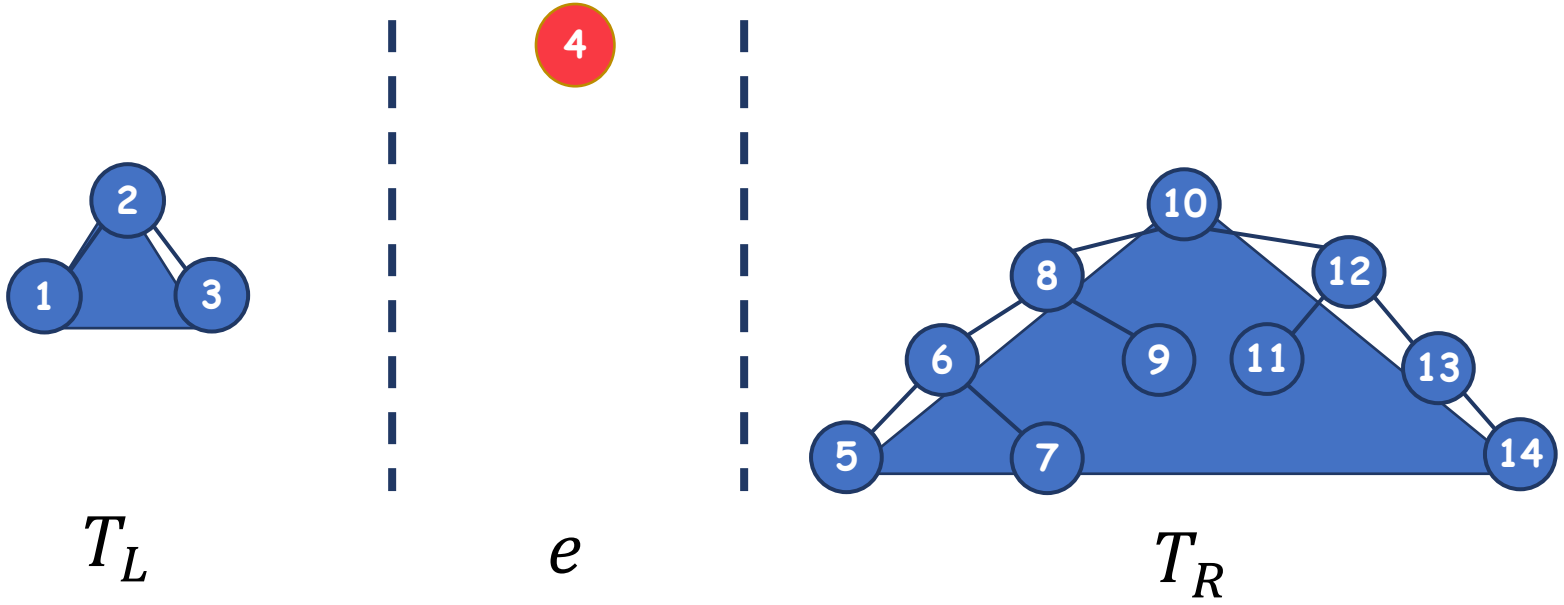
The Primitive `Join`

- $T = \text{Join}(T_L, e, T_R)$: T_L and T_R are two trees of a certain balancing scheme, e is an entry/node (the pivot).
- $\boldsymbol{T_L < e < T_R}$
- It returns a valid tree T, which is $T_L \cup \{e\} \cup T_R$



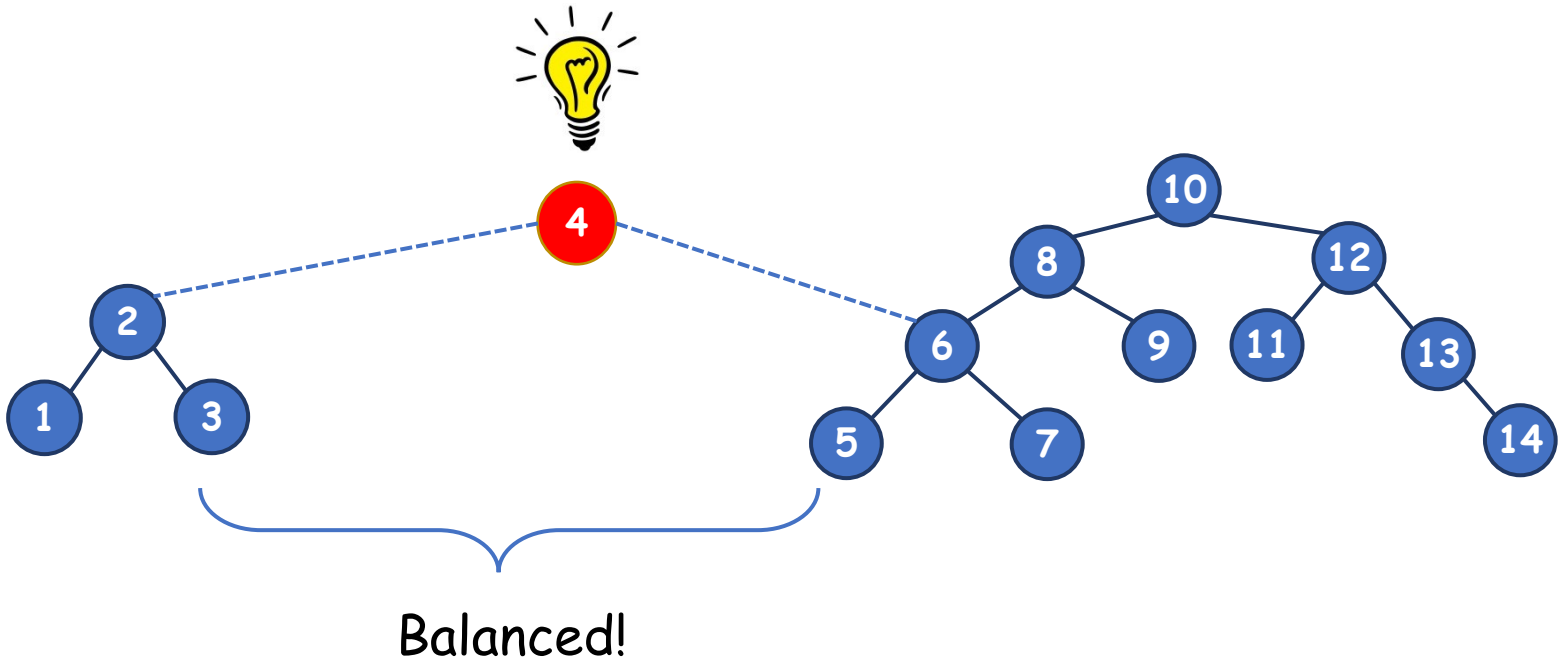
(Rebalance if necessary)

The Primitive `Join`



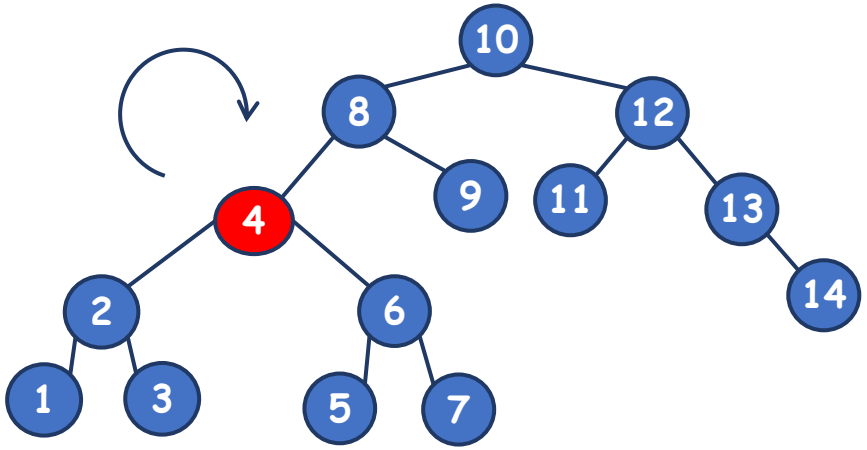
The Primitive `Join`

- Connect at a balancing point



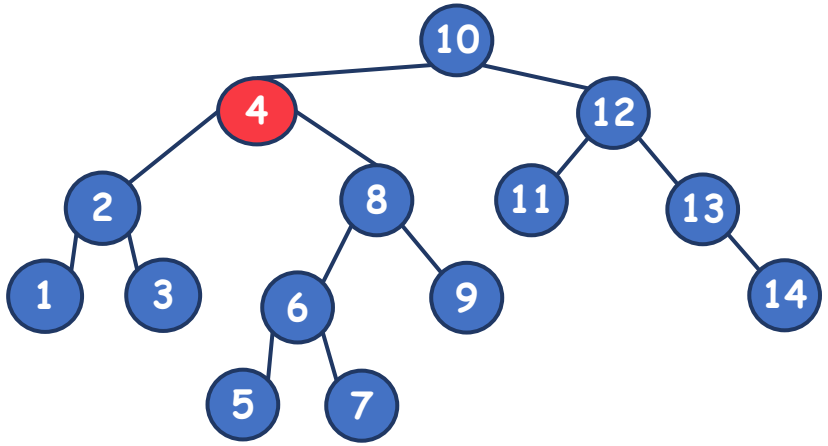
The Primitive `Join`

- Connect at a balancing point



The Primitive `Join`

- Connect at a balancing point
- Rebalance (specific to balancing schemes)
- Join algorithms for four balancing schemes and the cost bound [SPAA'16]



How does `Join` help?

1 Algorithms Using `Join`

- Generic across balancing schemes
- Highly-Parallel
- Theoretically efficient

2 Augmentation Using `Join`

- A unified framework for augmentation
- Model multiple applications

3 Persistence Using `Join`

- Multi-versioning on trees

Applications

Experiments

PART 1

Algorithms Using `Join`

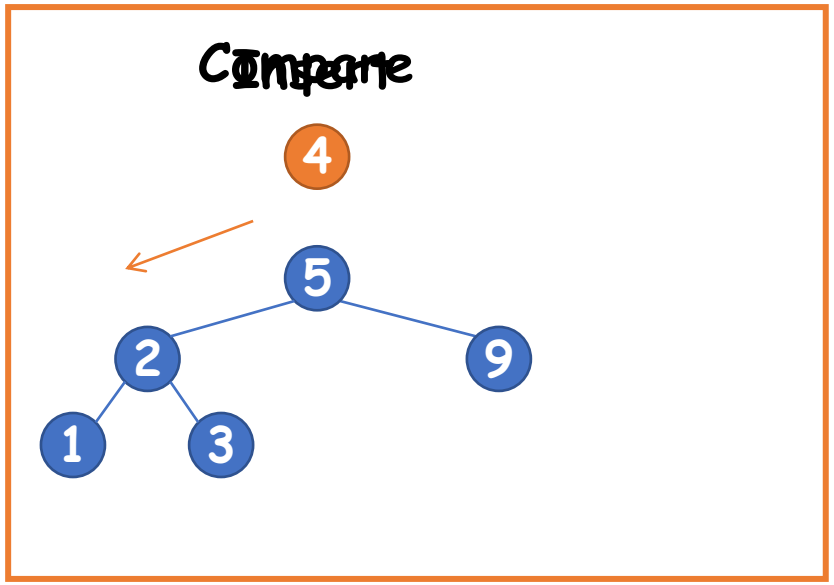
Generic algorithms across balancing schemes
Parallel algorithms using divide-and-conquer paradigm
Theoretically efficient

Join-based insertion

Join-based Algorithms: insertion

insert(T, k)

if $T = \emptyset$ **then return** `Singleton`(k)
else let $(L, k', R) = T$
if $k < k'$ **then**
    return `Join` (`Insert`(L, k),k',R)
else if $k > k(T)$ **then**
    return `Join` $(L, k',$ `Insert`$(R, k))$
else return T

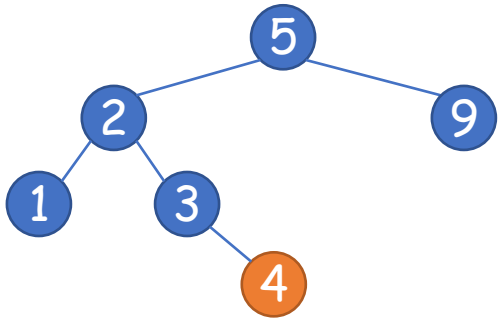


Join-based Algorithms: insertion

insert(T, k)

if $T = \emptyset$ **then return** `Singleton`(k)
else let $(L, k', R) = T$
if $k < k'$ **then**
    return `Join` (`Insert`(L, k),k',R)
else if $k > k(T)$ **then**
    return `Join` $(L, k',$ `Insert`$(R, k))$
else return T

Join results
(rebalanced may be required)

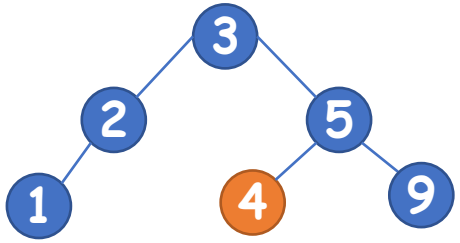


Join-based Algorithms: insertion

insert(T, k)

if $T = \emptyset$ **then return** `Singleton`(k)
else let $(L, k', R) = T$
if $k < k'$ **then**
    return `Join` (`Insert`(L, k),k',R)
else if $k > k(T)$ **then**
    return `Join` $(L, k',$ `Insert`$(R, k))$
else return T

Join results
(rebalanced may be required)



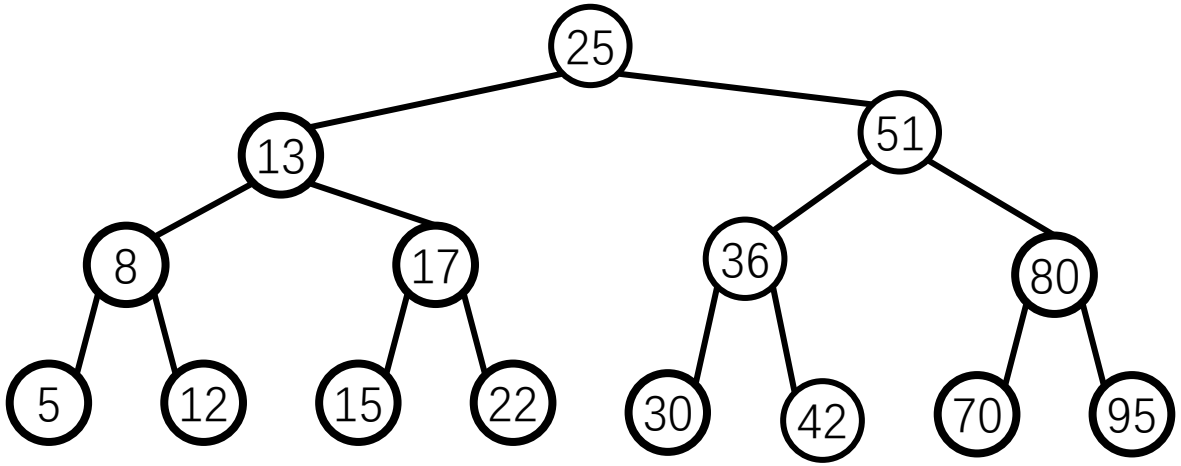
`Join` will rebalance for `Insert`!

$$O(\log n)$$

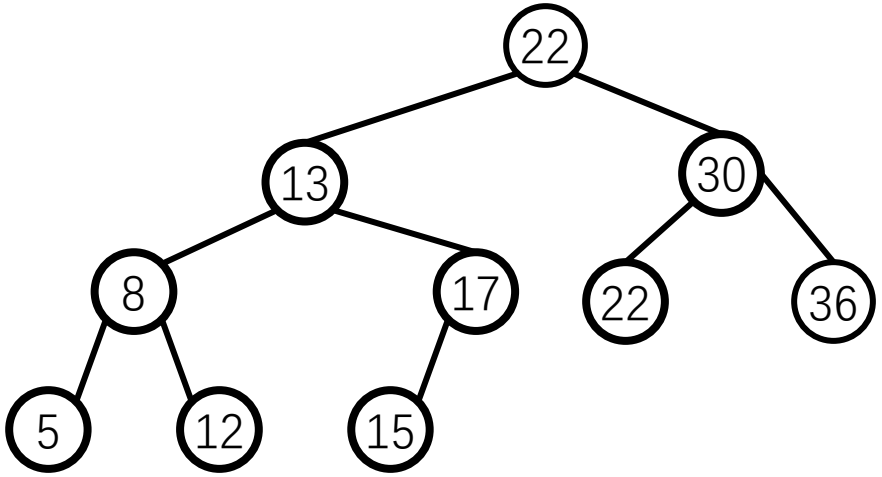
Join-based split and Join2

The Inverse of Join: Split

- $\langle T_L, b, T_R \rangle = \textit{Split}\,(T, k)$
  - T_L: all keys in $T < k$
  - T_R: all keys in $T > k$
  - A bit b indicating whether $k \in T$

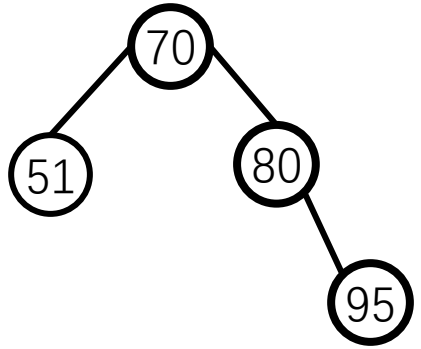


Split by 42



T_L

$b = true$

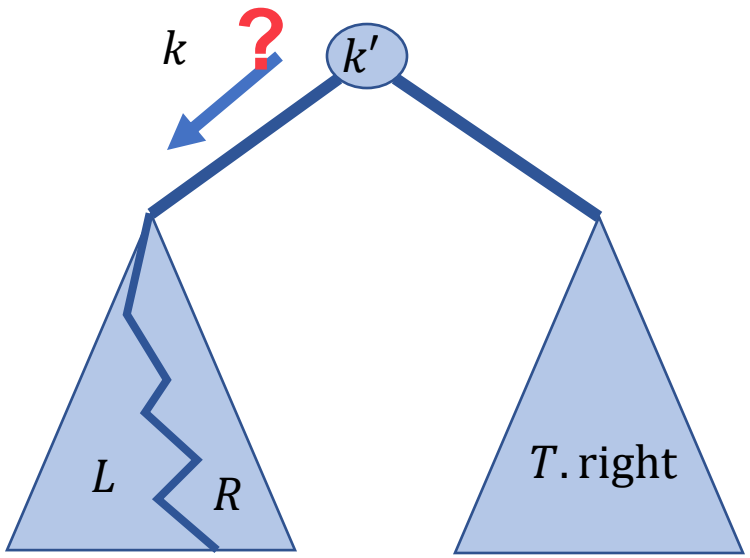


T_R

The Inverse of Join: Split

split(T, k)

if $T = \emptyset$ **then return** $(\emptyset, \mathbf{false}, \emptyset)$;
else {
  k' = key at the root of T
  if $k = k'$ **then** { // same key as the root
  $T_L = T.\text{left}$; $T_R = T.\text{right}$; flag = true}
  if $k < k'$ **then** { // split the left subtree
    $(L, \text{flag}, R) =$ **`split`**$(T.\text{left}, k)$;
    $T_L = L$;
    $T_R =$ **`Join`** $(R, k', T.\text{right})$; }
  if $k > k'$ **then** { /* symmetric*/ }
  return (T_L, flag, T_R)

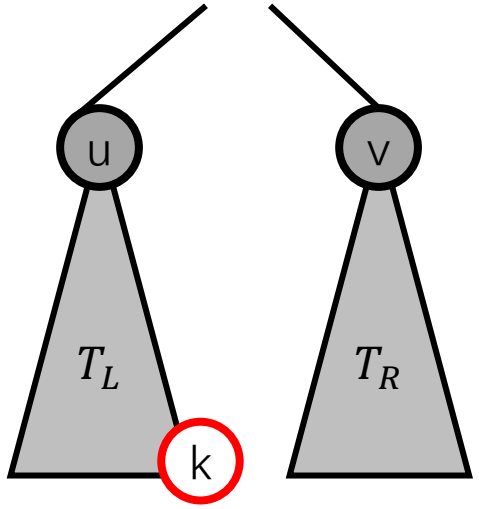


$T_L = L$

$T_R = \text{join}\ (R,\ k',\ T.right)$

Another helper function: join2

- join2(T_L, T_R)
- Similar to join, but without the middle key
- Can be done by first split out the last key in T_L, then use it to join the rest of T_L and T_R



join2(T_L, T_R) {
  (T_L', k) = split_last(T_L);
  return join(T_L', k, T_R);}

Other Join-based algorithms

BST Algorithms

- BST algorithms using divide-and-conquer scheme
  - Recursively deal with two subtrees (possibly in parallel)
  - Combine results of recursive calls and the root (e.g., using join or join2)
  - Usually gives polylogarithmic bounds on span

```
func(T, …) {
    if (T is empty)  return base_case;
    M = do_something(T.root);
    in parallel:
       L=func(T.left, …);
       R=func(T.right, …);
    return combine_results(L, R, M, …)
}
```

Get the maximum value

- In each node we store a key and a value. The nodes are sorted by the keys.

```
get_max(Tree T) {
    if (T is empty) return −∞;
    in parallel:
       L=get_max(T.left);
       R=get_max(T.right);
    return max(max(L, T.root.value), R);
```

$O(n)$ work and $O(\log n)$ span

Similar algorithm work on any map-reduce function

Map and reduce

- Maps each entry on the tree to a certain value using function *map*, then reduce all the mapped values using *reduce* (with identity *identity*).
- Assume map and reduce both have constant cost.

```
map_reduce(Tree T, function map, function reduce, value_type identity) {
    if (T is empty) return identity;
    M=map(t.root);
    in parallel:
       L=map_reduce(T.left, map, reduce, identity);
       R=map_reduce(T.right, map, reduce, identity);
    return reduce(reduce(L, M), R);
```

$O(n)$ work and $O(\log n)$ span

Filter

- Select all entries in the tree that satisfy function f
- Return a tree of all these entries

```
filter(Tree T, function f) {
    if (T is empty) return an empty tree;
    in parallel:
      L=filter(T.left, f);
      R=filter(T.right, f);
    if (f(T.root)) return join(L, T.root, R);
    else return join2(L, R); }
```

$O(n)$ work and $O(\log^2 n)$ span

Construction

```
T=build(Array A, int size) {
    A′=parallel_sort(A, size);
    return build_sorted(A′, s);
}

T=build_sorted(Arrary A, int start, int end) {
    if (start == end) return an empty tree;
    if (start == end-1) return singleton(A[start]);
    mid = (start+end)/2;
    in parallel:
      L = build_sorted(A, start, mid);
      R = build_sorted(A, mid+1, end);
    return join(L, A[mid], R);
```

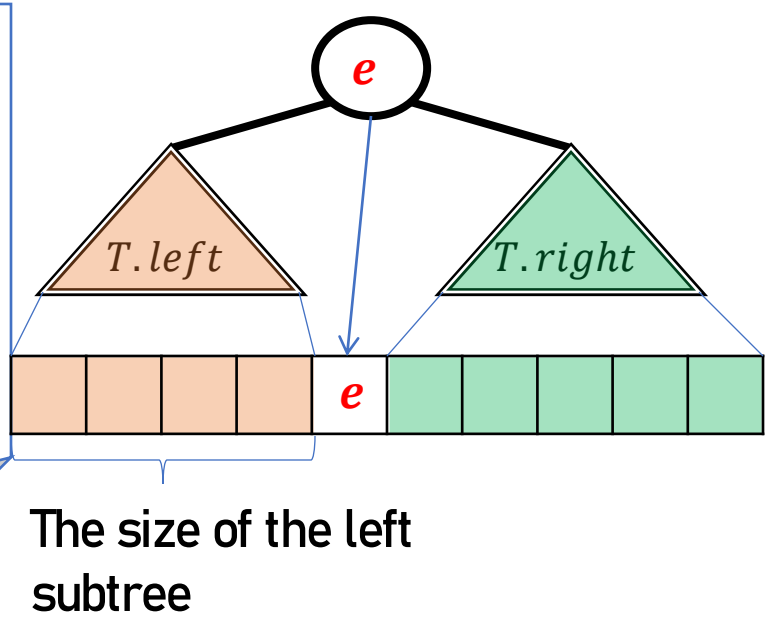
$O(n \log n)$ work and
$O(\log n)$ span,
bounded by the
sorting algorithm

$O(n)$ work and
$O(\log n)$ span

Output to array

- Output the entries in a tree T to an array in its in-order
- Assume each tree node stores its subtree size (an empty tree has size 0)

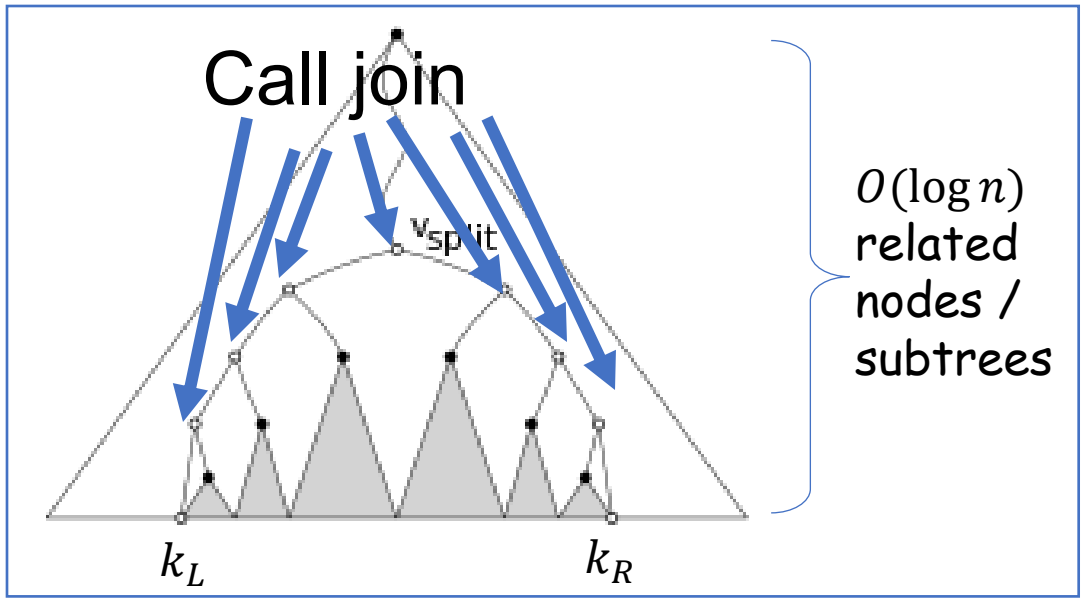
```
to_array(Tree T, array A, int offset) {
    if (T is empty) return;
    A[offset+T.left.size] = get_entry(T.root);
    in parallel:
       to_array(T.left, A, offset);
       to_array(T.right, A, offset+T.left.size()+1);
```



$O(n)$ work and $O(\log n)$ span

Range query (1D)

- Report all entries in key range $[k_L, k_R]$.
- Get a tree of them: $O(\log n)$ work and span
- Flatten them in an array: $O(k + \log n)$ work, $O(\log n)$ span for output size k



Equivalent to using two split algorithms

Range(T, k_L, k_R) {
  (t_1, b, t_2) = split(T, k_L);
  (t_3, b, t_4) = split(t_2, k_R);
  return t_3;
}

Join-based Algorithms: Union

- Input: T_1 and T_2 (size n and $m \leq n$)
- Output: T containing all elements in T_1 and T_2

- Can be used to combine a batch of elements to a tree

- The lower bound (of comparisons)

$$O\left(m \log\left(\frac{n}{m} + 1\right)\right)$$

- When $m = n$, it is $O(n)$
- When $n \gg m$, it is about $O(m \log n)$ (e.g., when $m = 1$, it is $O(\log n)$)

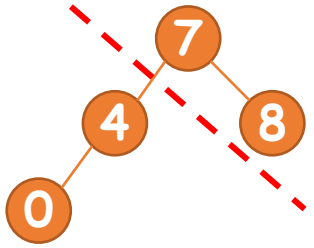
Join-based Algorithms: Union

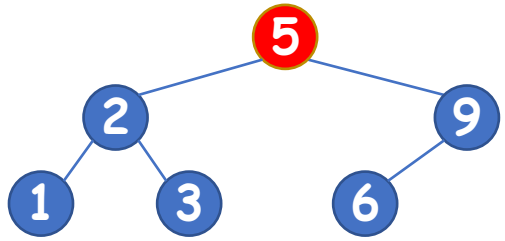
union(T_1, T_2)

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1

Base case

$(L_2, k_2, R_2) = extract(T_2)$
$(L_1, b, R_1) =$split(T_1, k_2)
In parallel:
$T_L =$Union(L_1, L_2)
$T_R =$Union(R_1, R_2)
return Join(T_L, k_2, T_R)





Join-based Algorithms: Union

union(T_1, T_2)

if $T_1 = \emptyset$ **then return** T_2

if $T_2 = \emptyset$ **then return** T_1

$(L_2, k_2, R_2) = extract(T_2)$

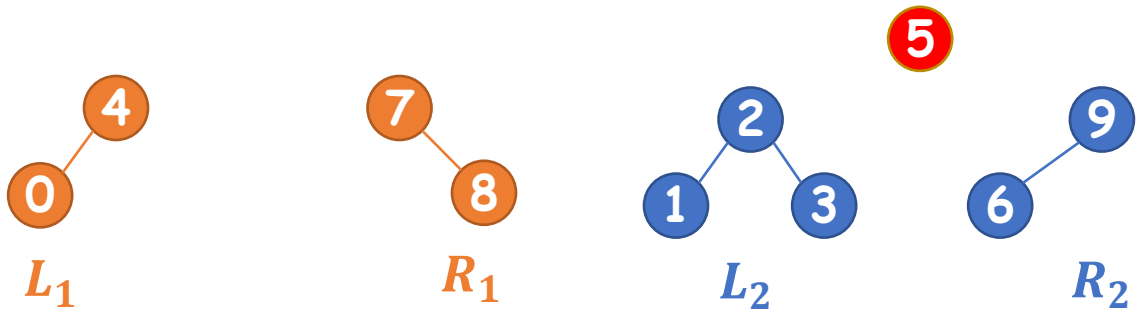
(L_1, b, R_1) =split(T_1, k_2)

In parallel:

T_L =Union(L_1, L_2)

T_R =Union(R_1, R_2)

return Join(T_L, k_2, T_R)

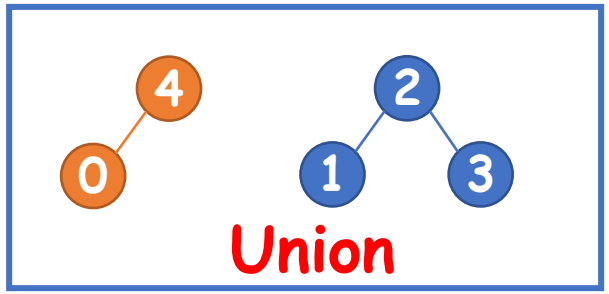


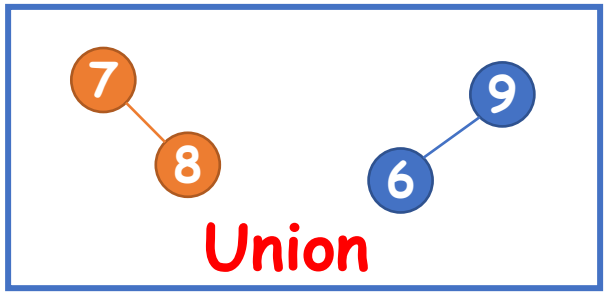
Join-based Algorithms: Union

union(T_1, T_2)

if $T_1 = \emptyset$ **then return** T_2
if $T_2 = \emptyset$ **then return** T_1
$(L_2, k_2, R_2) = extract(T_2)$
(L_1, b, R_1) =split(T_1, k_2)
In parallel:
T_L =Union(L_1, L_2)
T_R =Union(R_1, R_2)
return Join(T_L, k_2, T_R)

5





Join-based Algorithms: 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)

In parallel:

$T_L =$Union(L_1, L_2)

$T_R =$Union(R_1, R_2)

return Join(T_L, k_2, T_R)

Join-based Algorithms: Union

union(T_1, T_2)

if $T_1 = \emptyset$ **then return** T_2

if $T_2 = \emptyset$ **then return** T_1

$(L_2, k_2, R_2) = expose(T_2)$

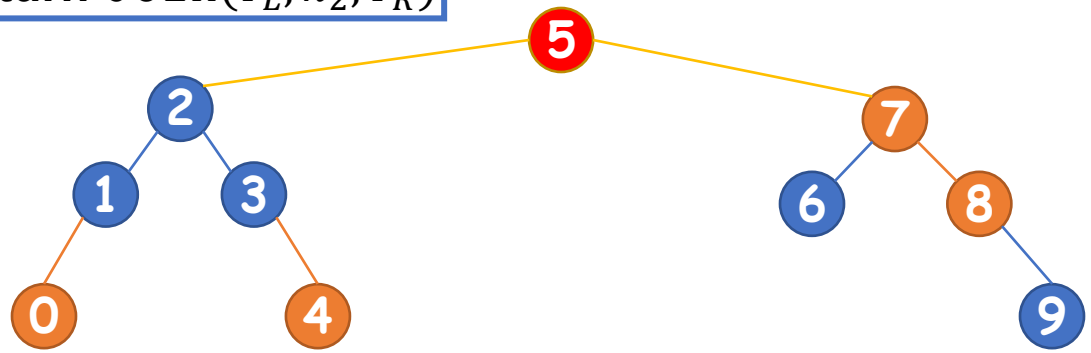
$(L_1, b, R_1) =$split(T_1, k_2)

In parallel:

$T_L =$Union(L_1, L_2)

$T_R =$Union(R_1, R_2)

return Join(T_L, k_2, T_R)



Similarly we can implement intersection and difference.

Join-based Algorithms: Union

Theorem 1. For AVL trees, red-black trees, weight-balance trees and treaps, the above algorithm of merging two balanced BSTs of sizes m and n ($m \leq n$) have $O\left(m \log\left(\frac{n}{m}+1\right)\right)$ work and $O(\log m \log n)$ span (in expectation for treaps).

- The bound also holds for intersection and difference

Join-based algorithms

- A wide variety of algorithms using *Join* are proved to be work-optimal and have polylogarithmic span

Functions	Work	Depth
`insert, delete, range, split, join2`	$O(\log n)$	$O(\log n)$
`union, intersection, difference`	$O\left(m \log\left(\frac{n}{m}+1\right)\right)$	$O(\log n \log m)$
`filter`	$O(n)$	$O(\log^2 n)$
`map_reduce`	$O(n)$	$O(\log n)$
`build`	$O(n \log n)$	$O(\log n)$
……		

Optimal **Polylog**

`Join` captures the intrinsic property of different balanced binary trees in **algorithms**.

Achieves speedup **50-90x** on 72 cores with hyperthreading

Outline

1 Algorithms Using `Join`

- Generic across balancing schemes
- Parallel using divide-and-conquer
- Simple
- Theoretically efficient
- Fast in practice

2 Augmentation Using `Join`

3 Persistence Using `Join`

PART 2

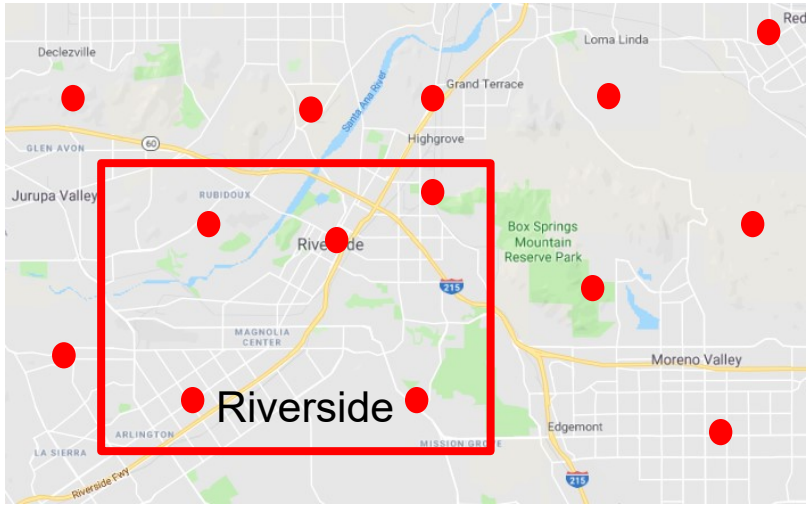
Augmentation Using `Join`

Augmented trees for fast range sum
The augmented map framework for applications

Augmentation for Range Queries

Geometric queries

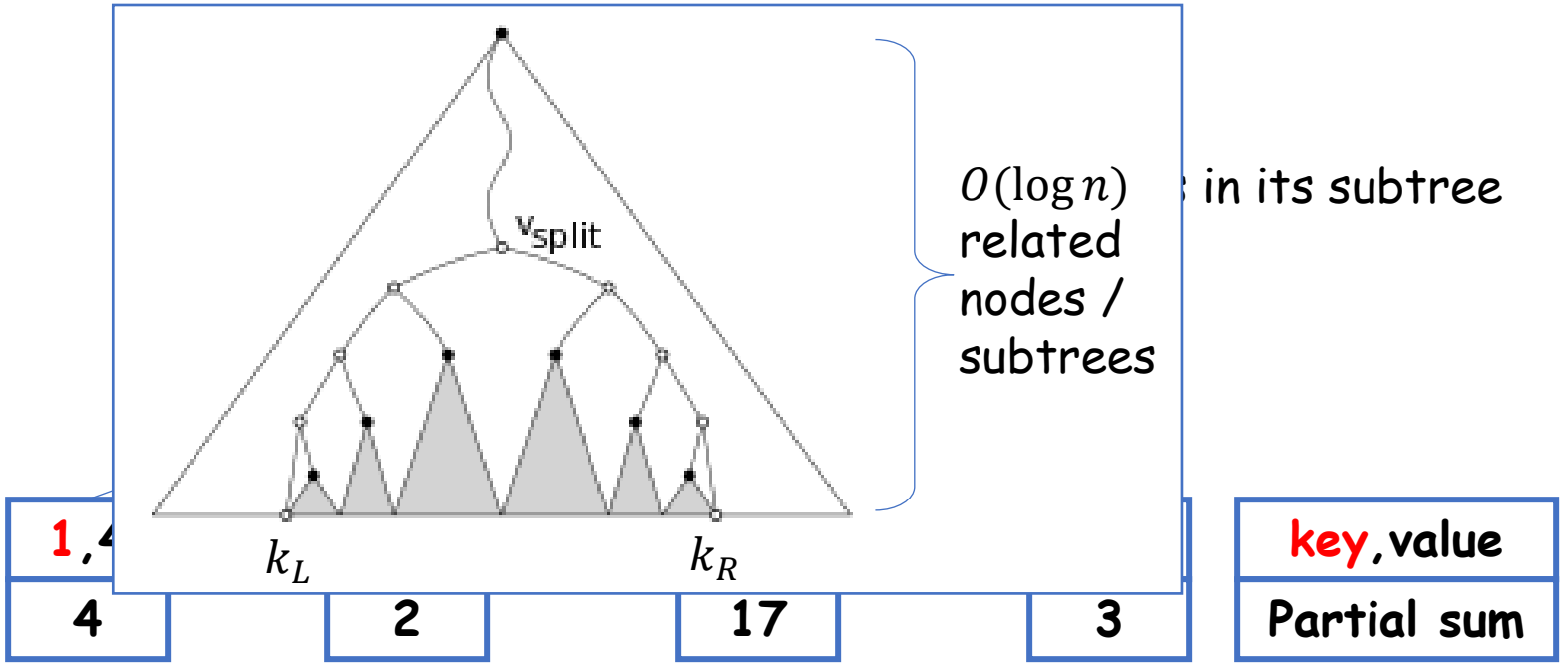
(A 2D range query)
Find average temperature
in Riverside area: 62 F



(A 2D range query)
Find all nearby Pokémon / Poke stops

Augmented Trees for 1D Range Query

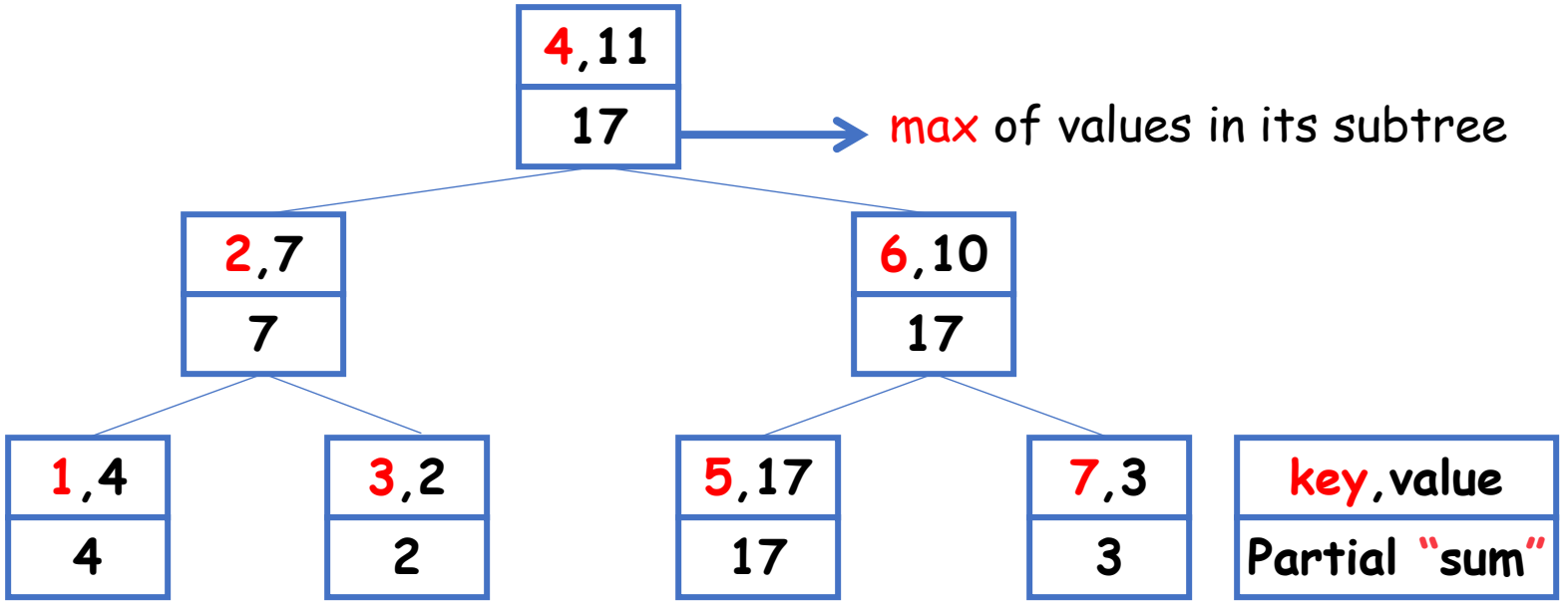
- Each tree node store some extra information about the *whole subtree* rooted at it (E.g., partial sum)



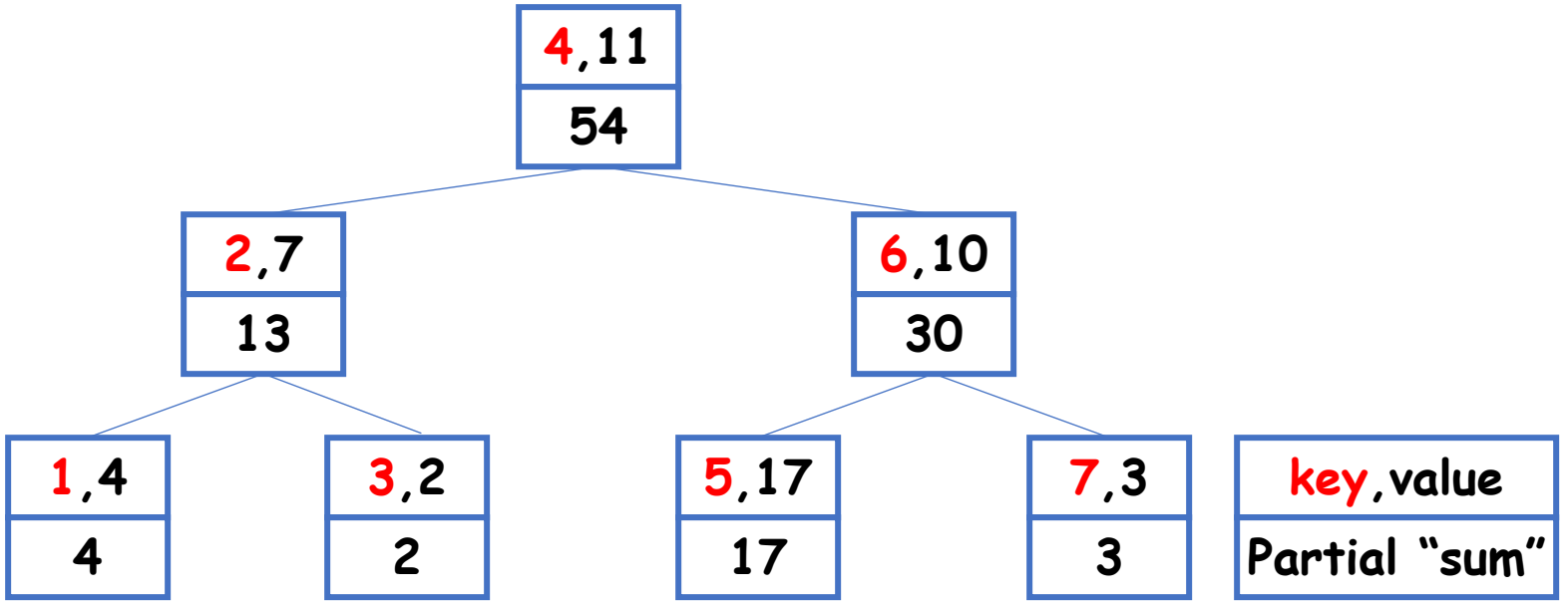
Range sum query (sum of values in a key range):
$O(\log n)$ time

Augmented Trees for 1D Range Query

- Different functionalities are achieved by different augmentations



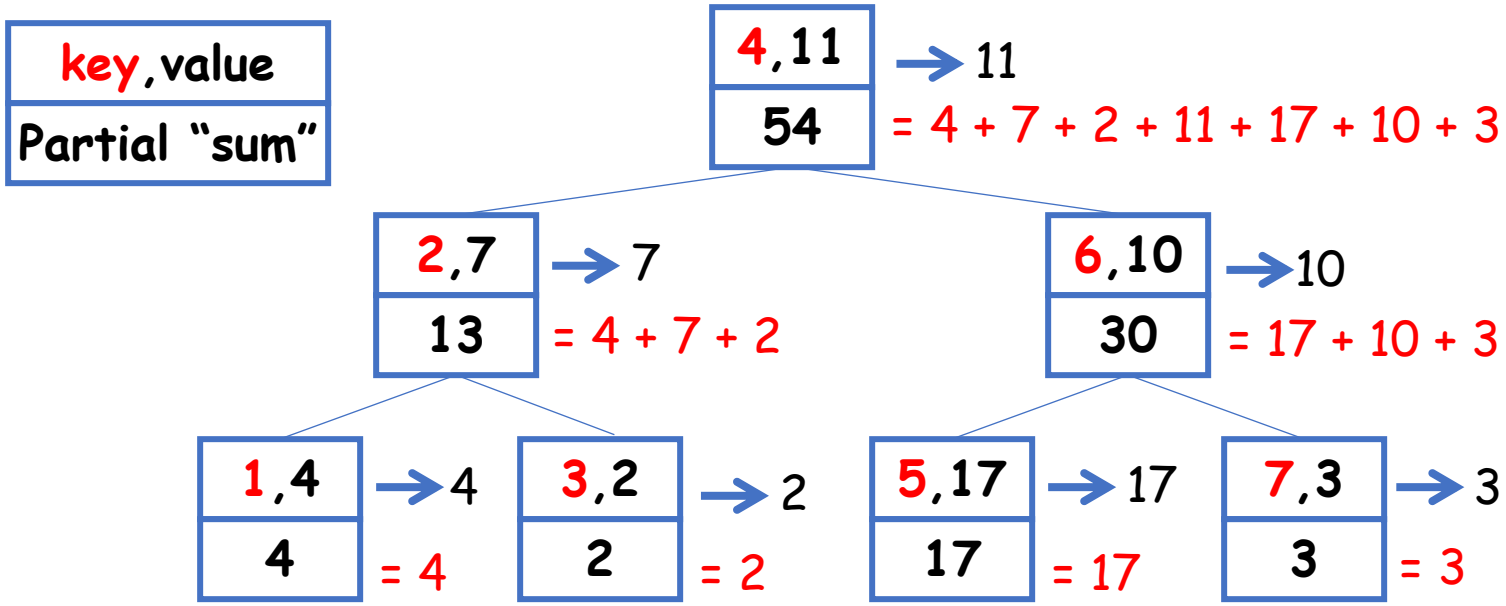
Augmentation – Formalization



- No formal definition for augmented trees…
- We give a definition with respect to ordered key-value pairs and a map-reduce operation
  - Each tree node stores a key-value pair ($K \times V$)
  - Each tree node also maintains some information about the whole subtree ($\in A$):

The *augmented value*

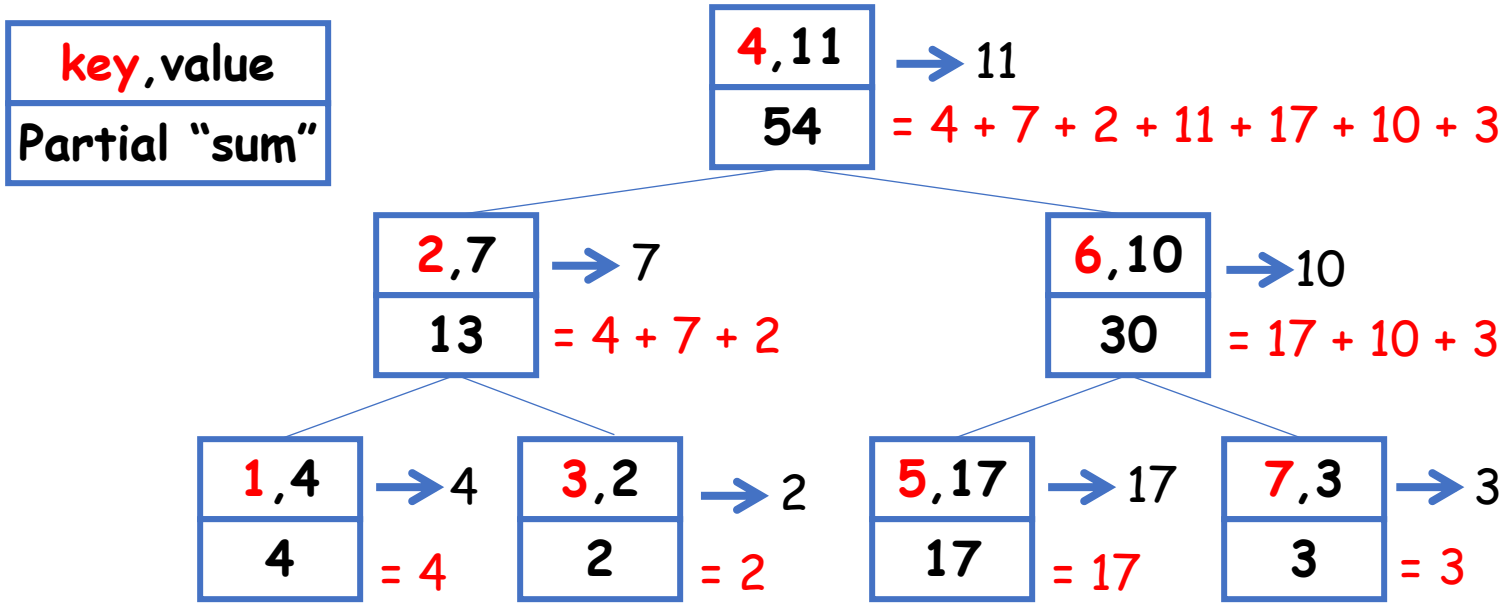
Augmentation – Formalization



The *augmented value*

- A "map" function $g: K \times V \mapsto A$ to map an entry to an augmented value
  g is $(k, v) \mapsto v$ in this example
- A "reduce" function $f: A \times A \mapsto A$ to reduce augmented values (associative, with identity I: (A, f, I) is a monoid)
  f is $+$ in this example

Augmentation – Formalization

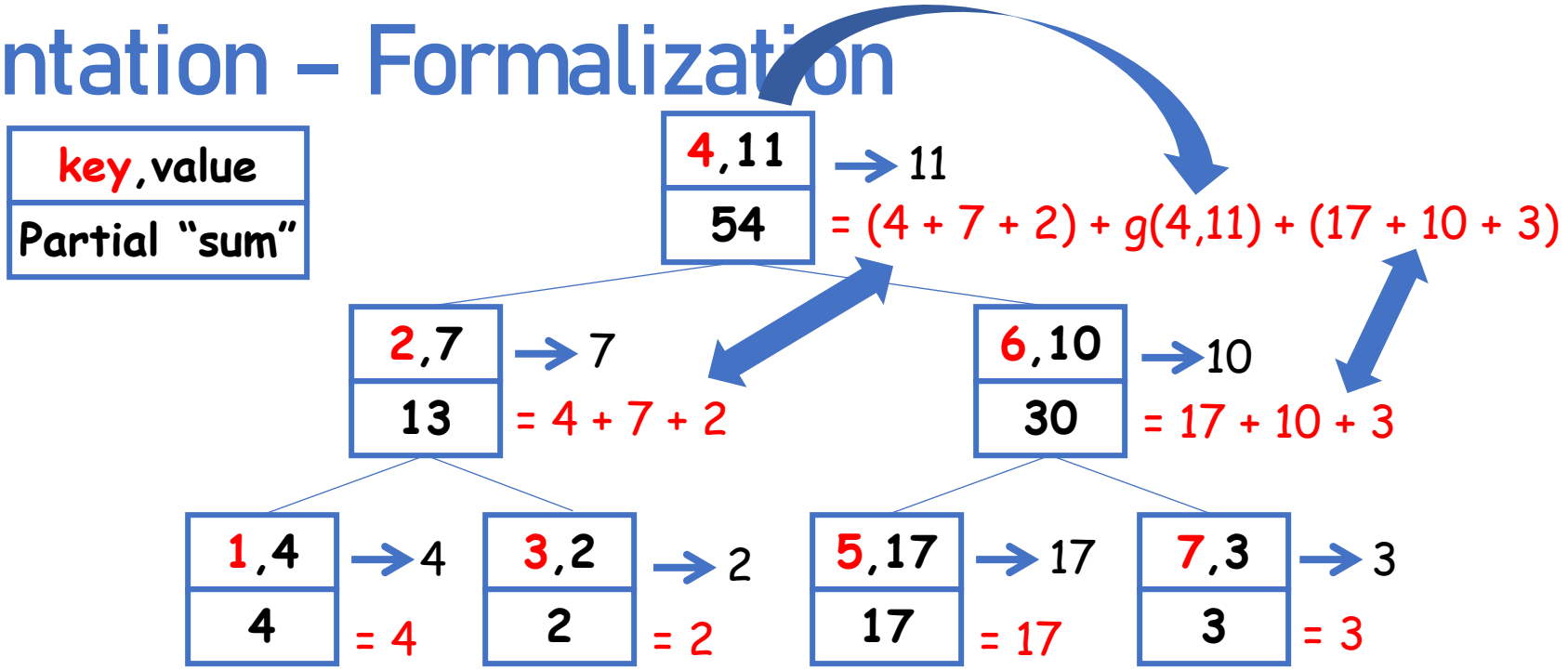


The *augmented value*

- A "map" function $g: K \times V \mapsto A$
- A "reduce" function $f: A \times A \mapsto A$, (A, f, I) is a monoid
- $a(u) = f\left(f\left(a(lc(u)), g(entry(u))\right), a(rc(u))\right)$

$a(u)$: the augmented value of node u, $entry(u)$: the entry stored in node u
$lc(u)$ and $rc(u)$: the left/right child of node u.

Augmentation – Formalization

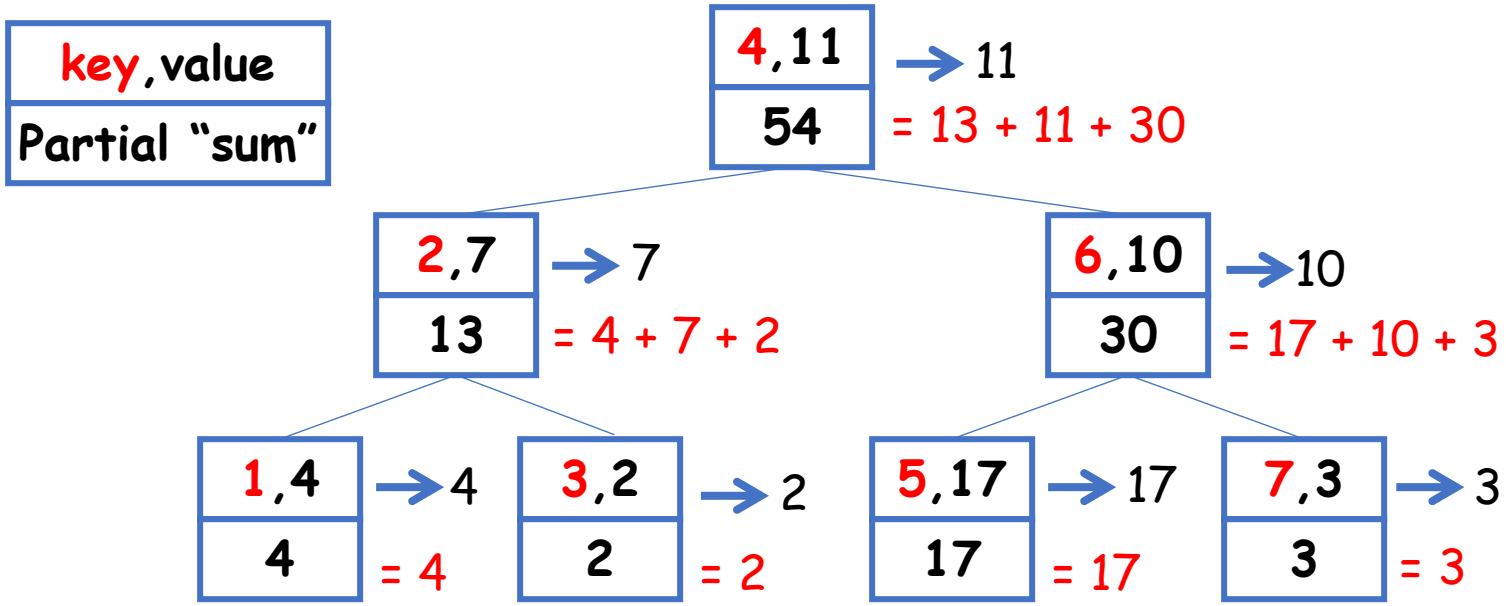


The *augmented value*

- A "map" function $g: K \times V \mapsto A$
- A "reduce" function $f: A \times A \mapsto A$, (A, f, I) is a monoid
- $a(u) = f\left(f\left(a(lc(u)), g(entry(u))\right), a(rc(u))\right)$

$a(u)$: the augmented value of node u, $entry(u)$: the entry stored in node u
$lc(u)$ and $rc(u)$: the left/right child of node u.

Augmentation – Formalization



The *augmented value*

- A "map" function $g: K \times V \mapsto A$
- A "reduce" function $f: A \times A \mapsto A, (A, f, I)$ is a monoid
- $a(u) = f\left(f\left(a(lc(u)), g(entry(u))\right), a(rc(u))\right)$
  - Update the augmented value only in Join
  - Does not affect the asymptotical cost if g and f are simple

Augmented trees

- Define keys and values
- Define what is the augmented value
- Define what is the map function
- Define what is the reduce function (and identity)

Augmented map

From Wikipedia, the free encyclopedia

In computer science, the **augmented map**[1] is an abstract data type (ADT) based on ordered maps, which associates each ordered map an **augmented value**. For an ordered map m with key type K, comparison function $<_K$ on K and value type V, the augmented value is defined based on two functions: a *base* function $g : K \times V \mapsto A$ and a *combine* function $f : A \times A \mapsto A$, where A is the type of the augmented value. The base function g converts a single entry in m to an augmented value, and the combine function f combines multiple augmented values. The combine function f is required to be associative and have an identity I (i.e., A, f, I forms a monoid). We extend the definition of the associative function f as follows:

$$f(\emptyset) = I$$
$$f(a) = a$$
$$f(a_1, a_2, \ldots, a_n) = f(f(a_1, a_2, \ldots, a_{n-1}), a_n)$$

Then the augmented value of an ordered map $m = \{(k_1, v_1), (k_2, v_2), \ldots (k_n, v_n)\}$ is defined as follows:

$$A(m) = f(g(k_1, v_1), g(k_2, v_2), \ldots, g(k_n, v_n))$$

Accordingly, an augmented map can be formally defined as a seven-tuple $\mathbb{AM}(K, <_K, V, A, g, f, I)$. For example, an augmented map with integral keys and values, on which the augmented value is defined as the sum of all values in the map, is defined as:

$$M_1 = \mathbb{AM}(\mathbb{Z}, <_{\mathbb{Z}}, \mathbb{Z}, \mathbb{Z}, (k, v) \mapsto v, +_{\mathbb{Z}}, 0)$$

The Wikipedia page of the augmented map defined by us [PPoPP'18], which is an abstract data type (ADT) that extends the augmented trees

Augmentation: Applications

- Can be applied to a variety of applications:

	$\mathbb{AM}(K,$	$<_K,$	$V,$	$A,$	$g,$	$f,$	$I)$
❑ **Range Sum:**							
	$S = \mathbb{AM}(\mathbb{Z},$	$<_{\mathbb{Z}},$	$\mathbb{Z},$	$\mathbb{Z},$	$(k,v) \mapsto v,$	$+_{\mathbb{Z}},$	$0)$
❑ **1D stabbing query (can yield an *interval tree*):**							
	$T = \mathbb{AM}(\mathbb{R},$	$<_{\mathbb{R}},$	$\mathbb{R},$	$\mathbb{R},$	$(k,v) \mapsto v,$	$max,$	$-\infty)$
❑ **2D range query (can yield a *range tree*):**							
	$R_I = \mathbb{AM}($ $X \times Y,$	$<_Y,$	$W,$	$W,$	$(k,v) \mapsto v,$	$+_W,$	$0_W)$
	$R_O = \mathbb{AM}($ $X \times Y,$	$<_X,$	$W,$	$R_I,$	R_I.singleton	R_I.union,	R_I.empty)
❑ **Document searching:**							
	$M_I = \mathbb{AM}($ $D,$	$<_D,$	$W,$	$W,$	$(k,v) \mapsto v,$	$max,$	$0)$
	$M_O = \mathbb{AM}($ $T,$	$<_T,$	$M_I,$	-,	-,	-,	-)

- Also, many tree-based geometric problems like segment queries, rectangle queries, …

Code for the Interval Tree

```
struct interval_map {
  using interval = pair<int, int>;
  struct entry {
    using key_t = int;
    static bool comp(key_t a, key_t b) {
      return a < b;}
    using val_t = int;
    using aug_t = int;
    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;}
    static aug_t I() { return 0;}
  };
  using aug_map = aug_map<entry>;
  aug_map m;

  interval_map(interval* A, int n) {
    m = aug_map(A,A+n); }

  bool stabbing(int p) {
    return (m.aug_left(p) > p);}};
```

g and f

Construction

Query

Branch: master ▾ PAM / interval / intervalTree.h

syhlalala intervalTree.h

1 contributor

55 lines (42 sloc) | 1.14 KB

Interval tree, 53 lines

Branch: master ▾ PAM / range_query / range_tree.h

syhlalala changes

1 contributor

176 lines (147 sloc) | 4.99 KB

Range tree, 176 lines

Branch: master ▾ PAM / index / index.h

gblelloch changes

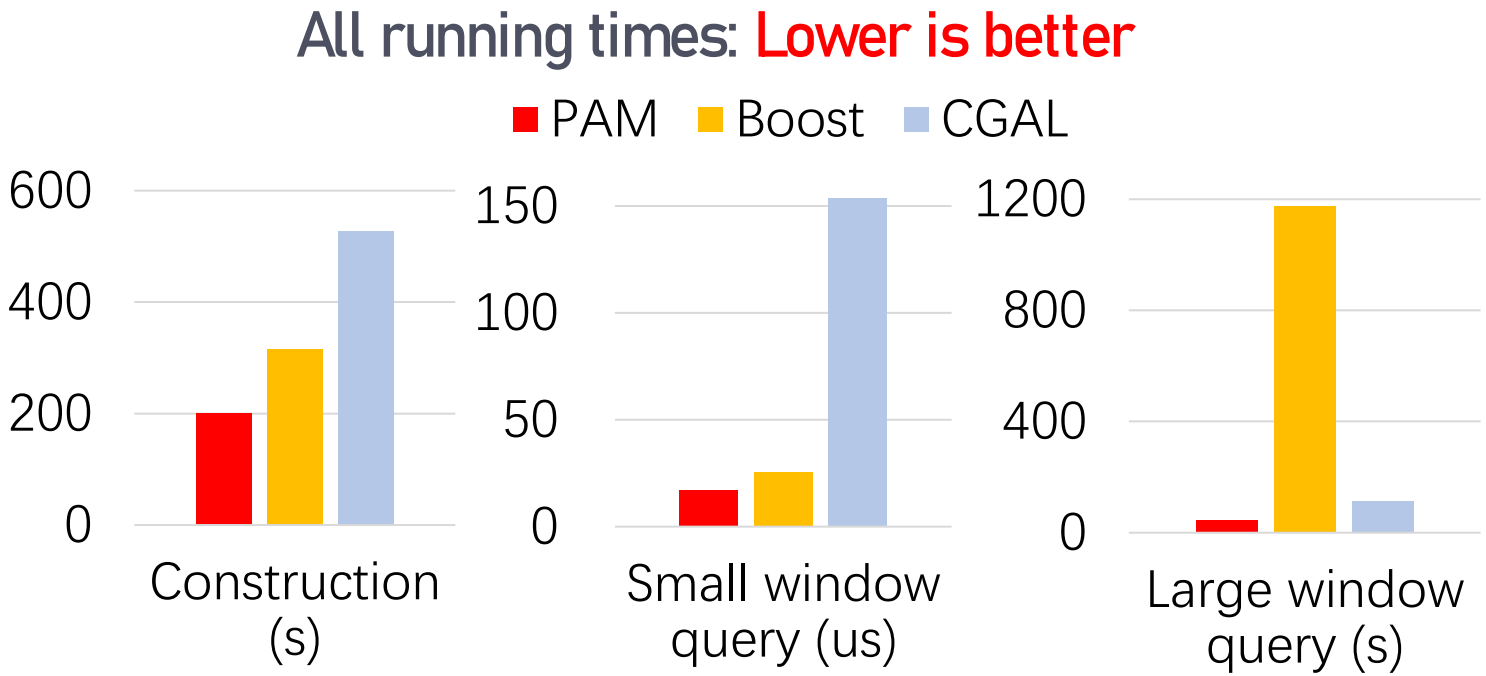
2 contributors

98 lines (81 sloc) | 2.68 KB

Document searching, 98 lines

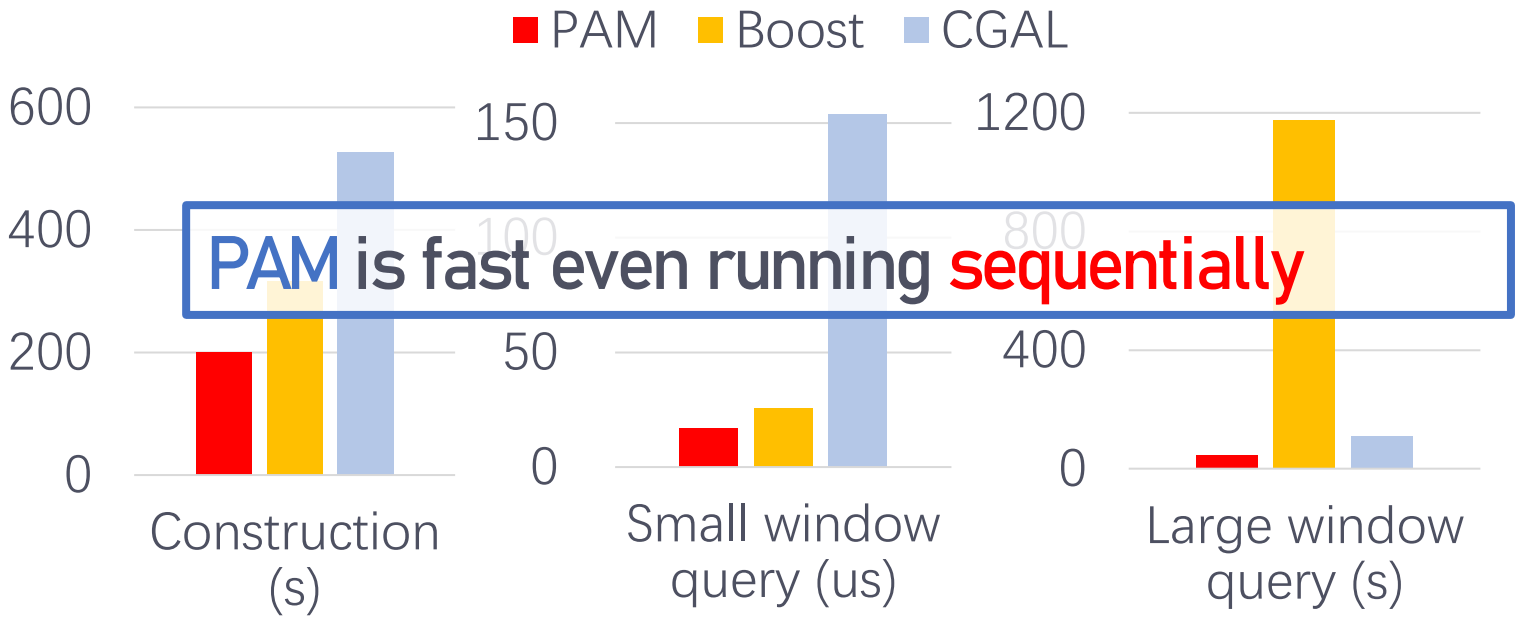
Range Queries – Sequential and Parallel [PPoPP'18, ALENEX'19]

- 10^8 input points
- Run PAM directly on one core
- Compare to range tree in CGAL and R-tree in Boost
- Both are sequential libraries



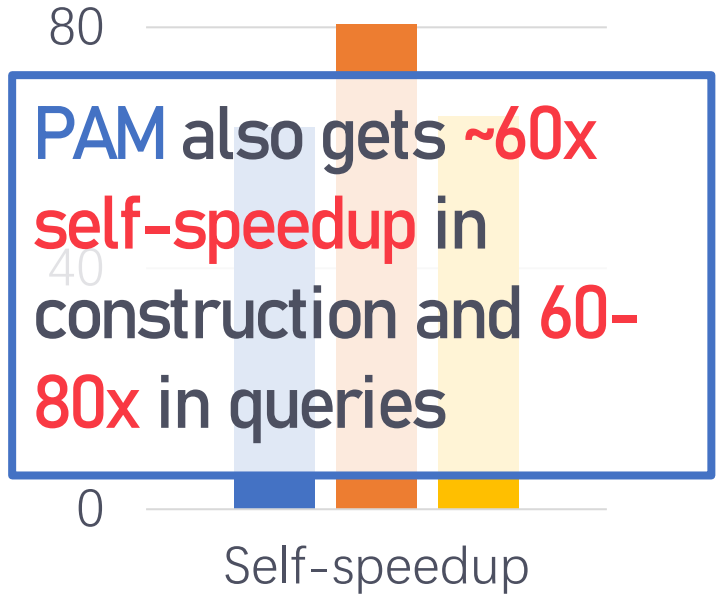
Range Queries – Sequential and Parallel [PPoPP'18, ALENEX'19]

All running times: Lower is better

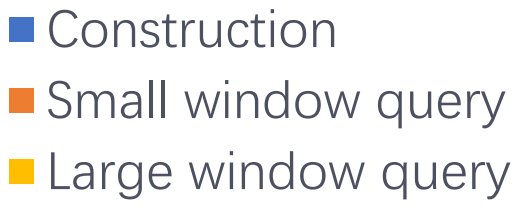


PAM is fast even running sequentially

Self-speedup of PAM
72 cores, 144 threads
Higher is better



PAM also gets ~60x self-speedup in construction and 60-80x in queries



Construction

2x Faster than CGAL

1.5x Faster than Boost

Query

2-9x Faster than CGAL

1.4-26x Faster than Boost

Applications using augmentation

- 1D stabbing query
- 2D range query, segment query, rectangle query
- 2D sweepline algorithms
- Inverted index searching

- Example code for all applications available on Github
  - https://github.com/cmuparlay/PAM
- Algorithms and more experiments available:
  - PAM: Parallel Augmented Maps, Yihan Sun, Daniel Ferizovic and Guy Blelloch, ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), 2018
  - Parallel Range, Segment and Rectangle Queries with Augmented Maps, Yihan Sun and Guy E. Blelloch, Algorithm Engineering and Experiments (ALENEX), 2019

Outline

1 Algorithms Using Join

2 Augmentation Using Join

- Formalize augmented trees: implemented by just join
- Multiple applications: range-sum, stabbing queries, range queries, inverted index searching
  - Simple code
  - Parallel solution
  - Fast in practice

3 Persistence Using Join

PART 3

Persistence Using `Join`

Persistent algorithms
Multi-version concurrency control (MVCC)

What Are Persistence and MVCC?

- Persistence [DSST'86]: for data structures
  - Preserves the previous version of itself
  - Always yields a new version when being updated
- Multi-version Concurrency Control (MVCC): for databases
  - Let write transactions create new versions
  - Let ongoing queries work on old versions

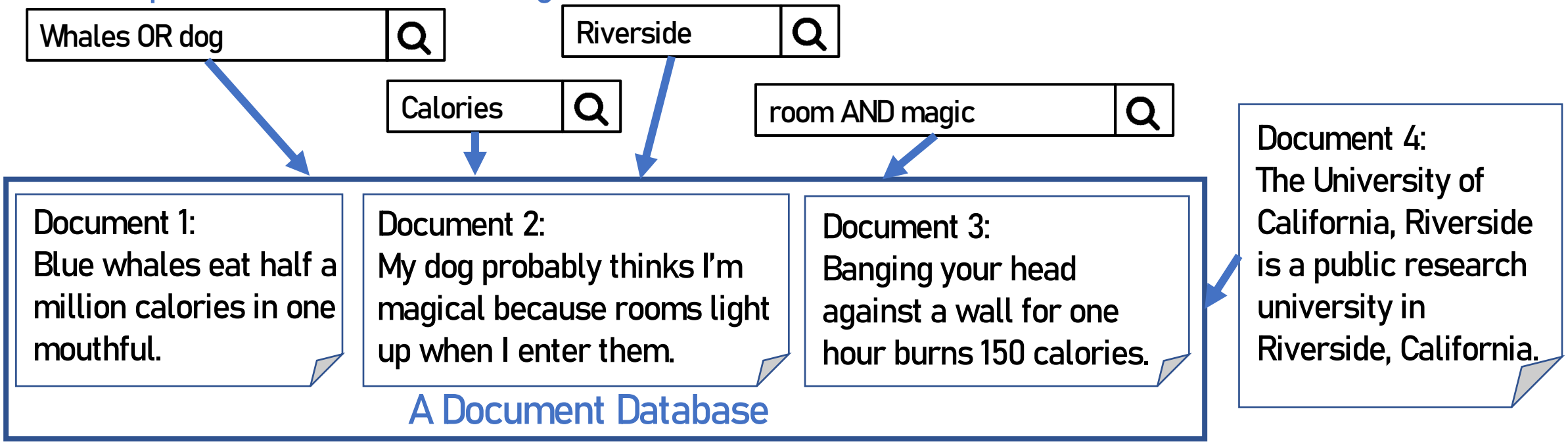
$$T_2 = T.\text{insert}(k)$$

Why Persistence and MVCC?

- To guarantee concurrent updates and queries to work correctly and efficiently
  - Queries work on a consistent version
  - Writers/readers do not block each other

Why Persistence and MVCC?

An example of a document search engine.



For end-user experience:

- Queries shouldn't be delayed by updates
- Queries must be done on a consistent version of database

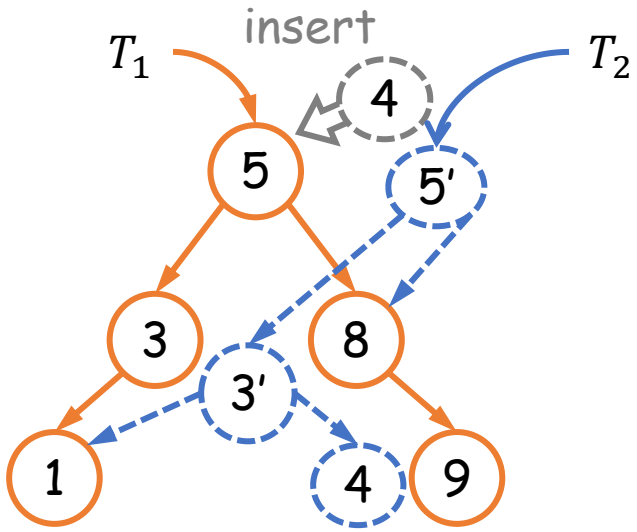
Generally useful for any database systems with concurrent updates and queries.

Hybrid Transactional and Analytical Processing (HTAP) Database System

Persistence Using Join

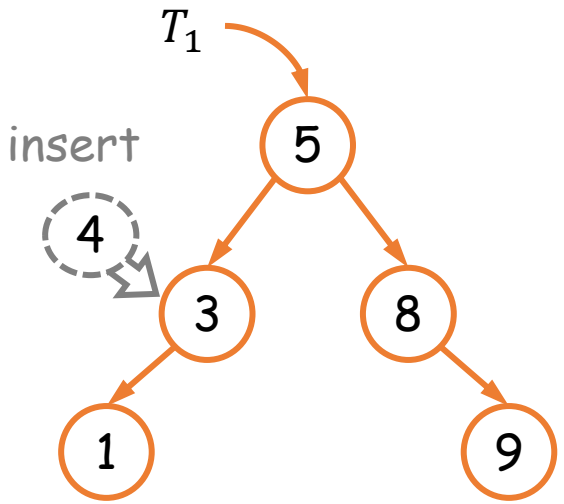
- Path-copying: copy the affected path on trees
- Copying occur only in Join!

$$T_2 = T_1.\text{insert}(4)$$



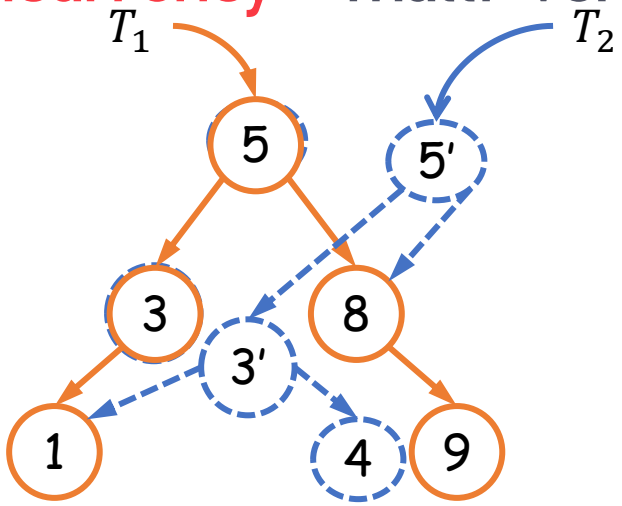
Persistence Using Join

- Path-copying: copy the affected path on trees
- Copying occur only in Join!



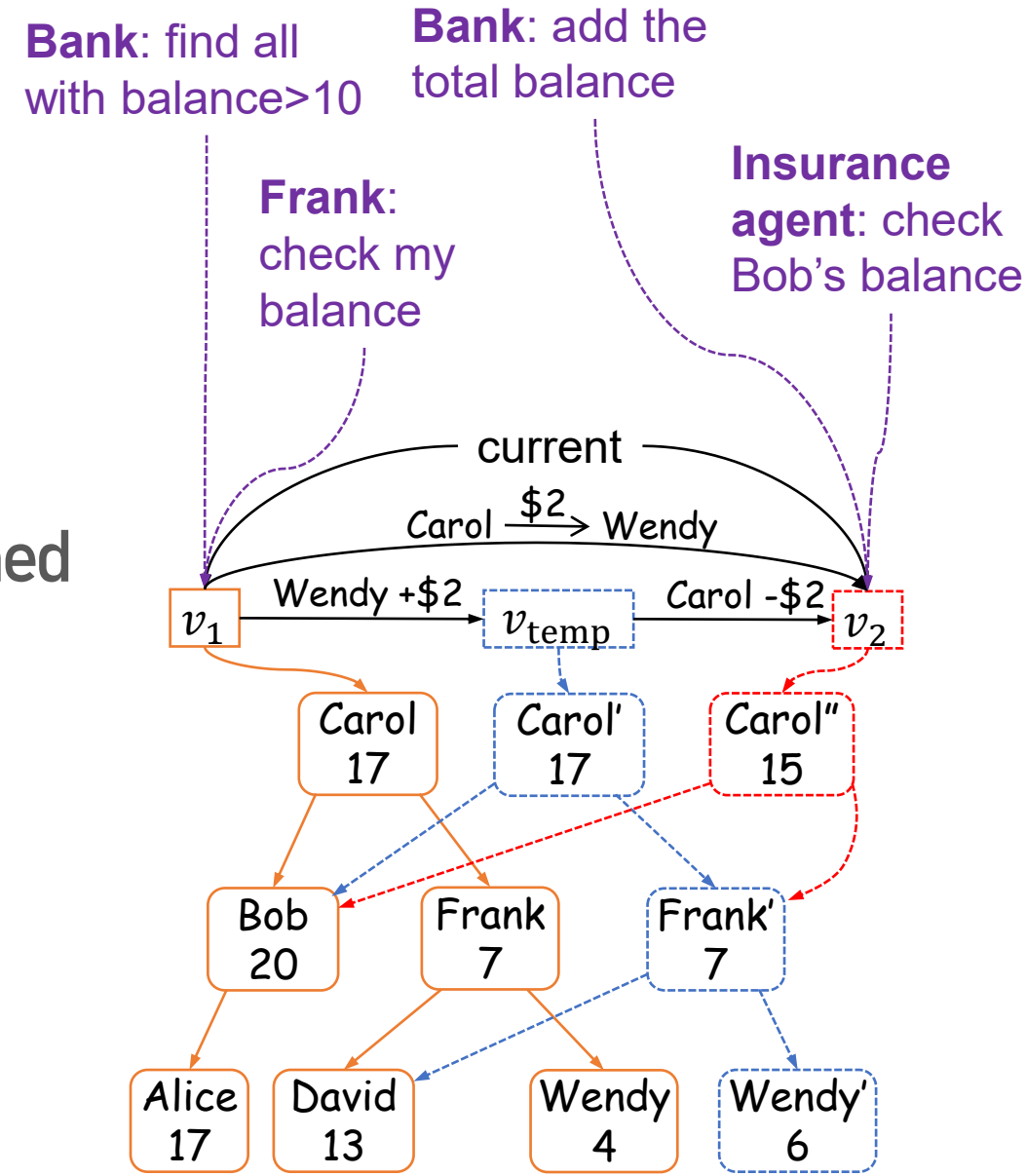
Persistence Using Join

- Path-copying: copy the affected path on trees
- Copying occur only in Join!
- Always copy the middle node
- All the other parts in the algorithm remain unchanged
- No extra cost in time asymptotically, small overhead in space
- Safe for concurrency – multi-version concurrency control (MVCC)



Transactions using Multi-version Concurrency Control (MVCC)

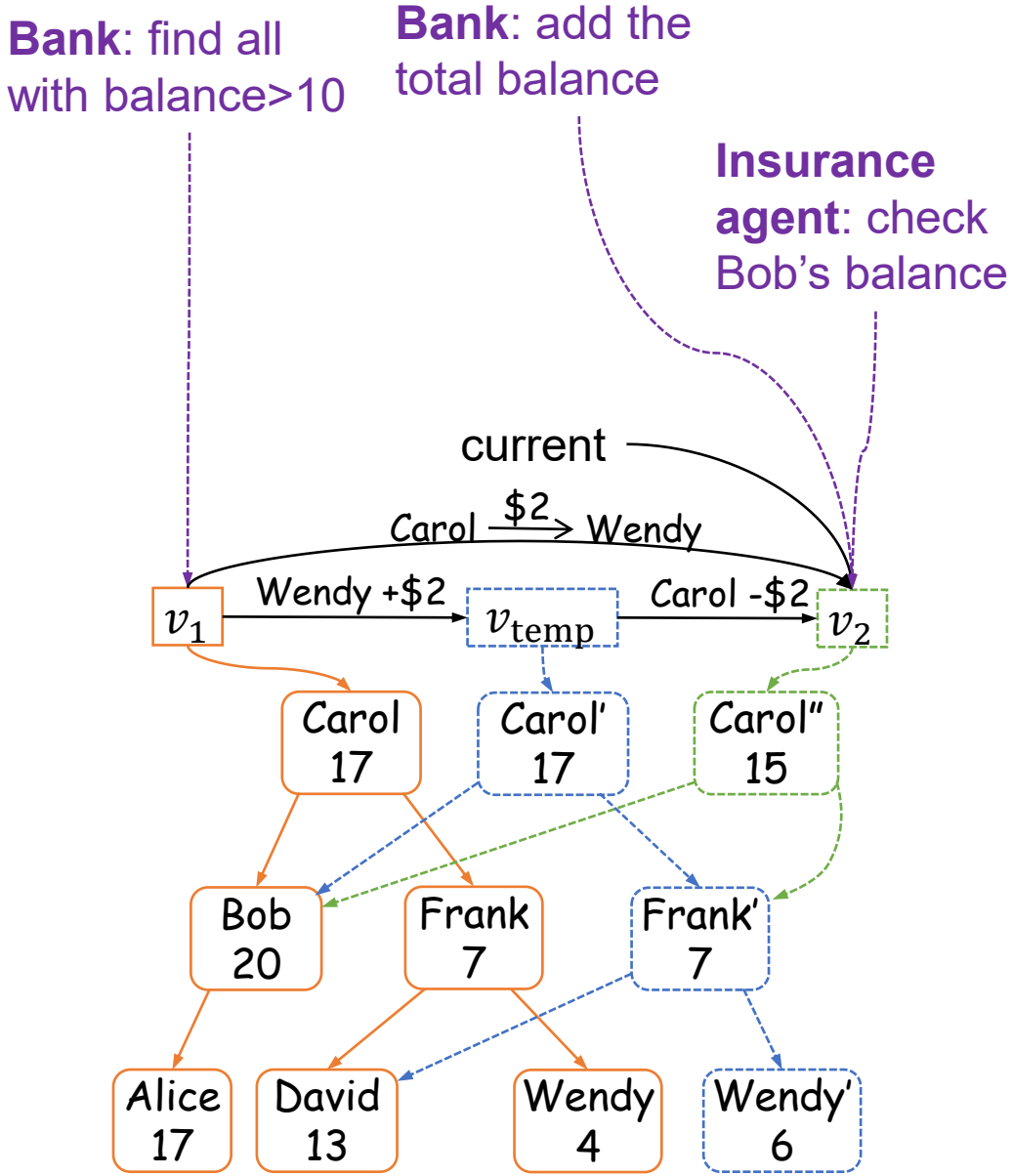
- Lock-free **atomic** updates ☺
  - A series of operations
  - A bulk of operations (e.g., union)
- Easy **roll-back** ☺
- Do not affect other **concurrent** operations ☺
- Any operation works on as if a single-versioned tree with **no extra (asymptotical) cost** ☺



Transactions using Multi-version Concurrency Control (MVCC)

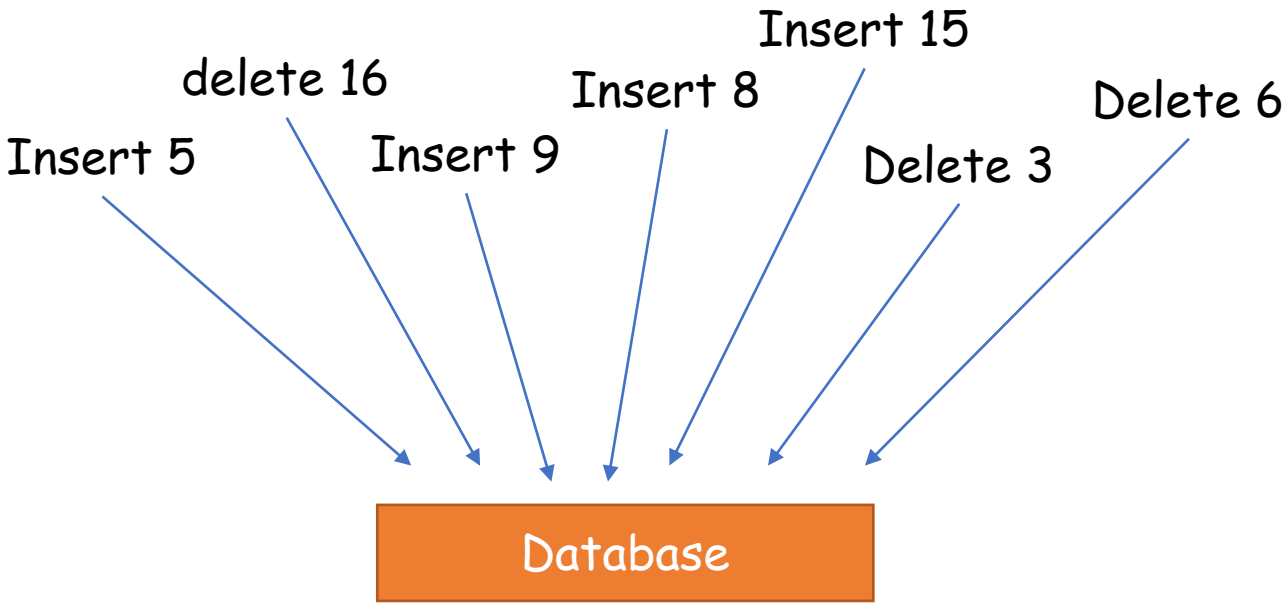
- Lock-free atomic updates ☺
  - A series of operations
  - A bulk of operations (e.g., union)
- Easy roll-back ☺
- Do not affect other concurrent operations ☺
- Any operation works on as if a single-versioned tree with no extra (asymptotical) cost ☺

- Concurrent writes? ☹
  - Concurrent transactions work on snapshots
  - They don't come into effect on the same tree?
- Useless old nodes? ☹
  - Out-of-date nodes should be collected in time



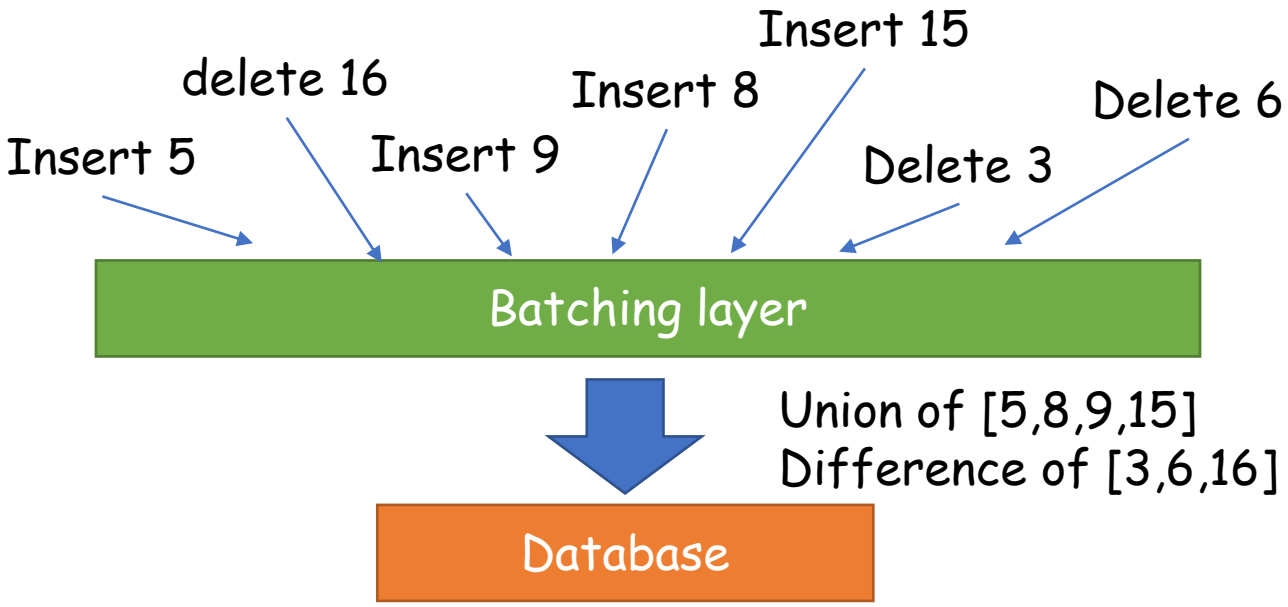
Batching

- Collect all concurrent writes can commit using a single writer once a while



Batching

- Collect all concurrent writes can commit using a single writer once a while

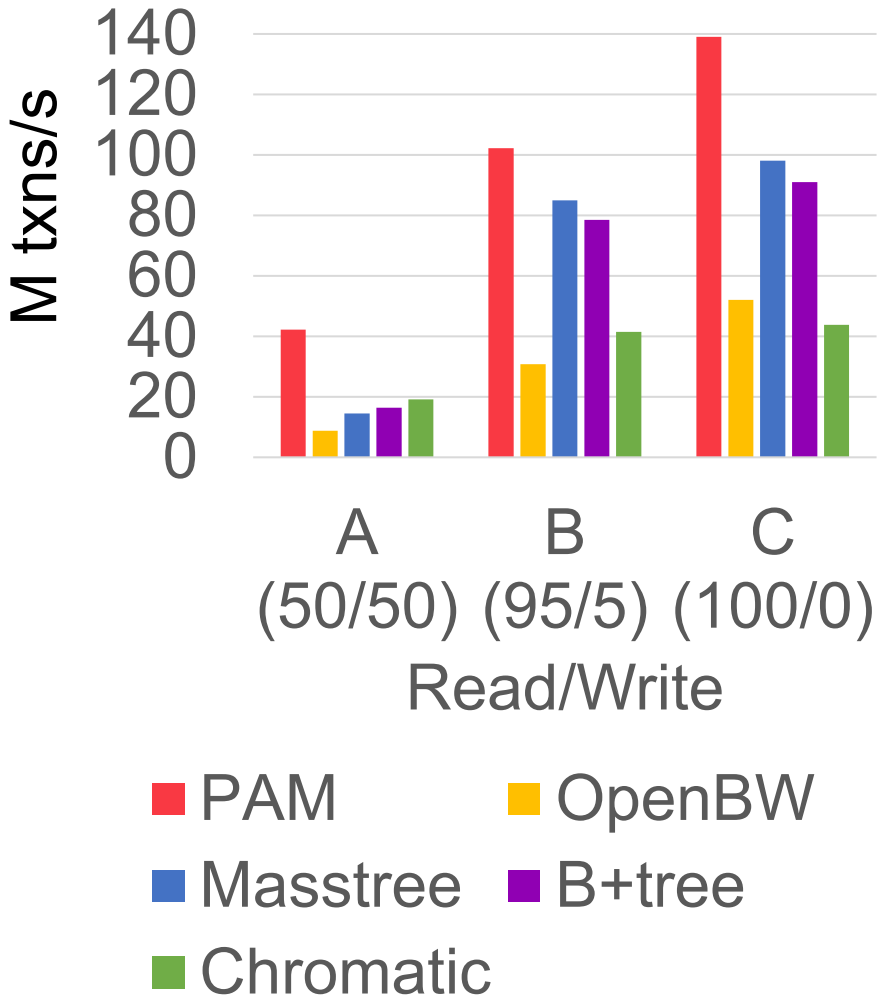


Compare to Concurrent Data Structures [SPAA'19]

- Compare with concurrent data structures
  - Skiplist, OpenBW tree [Wang et al.], Masstree [Mao et al.], B+tree [Wang et al.], Chromatic tree [Brown et al.]
- Test on Yahoo! Cloud Serving Benchmark (YCSB)
  - Streams of updates and queries (insertion/update/searching)
- Initial database size: 5×10^7, transactions: 10^7
- Delay within 50ms

Compare to Concurrent Data Structures [SPAA'19]

Throughput: Higher is better

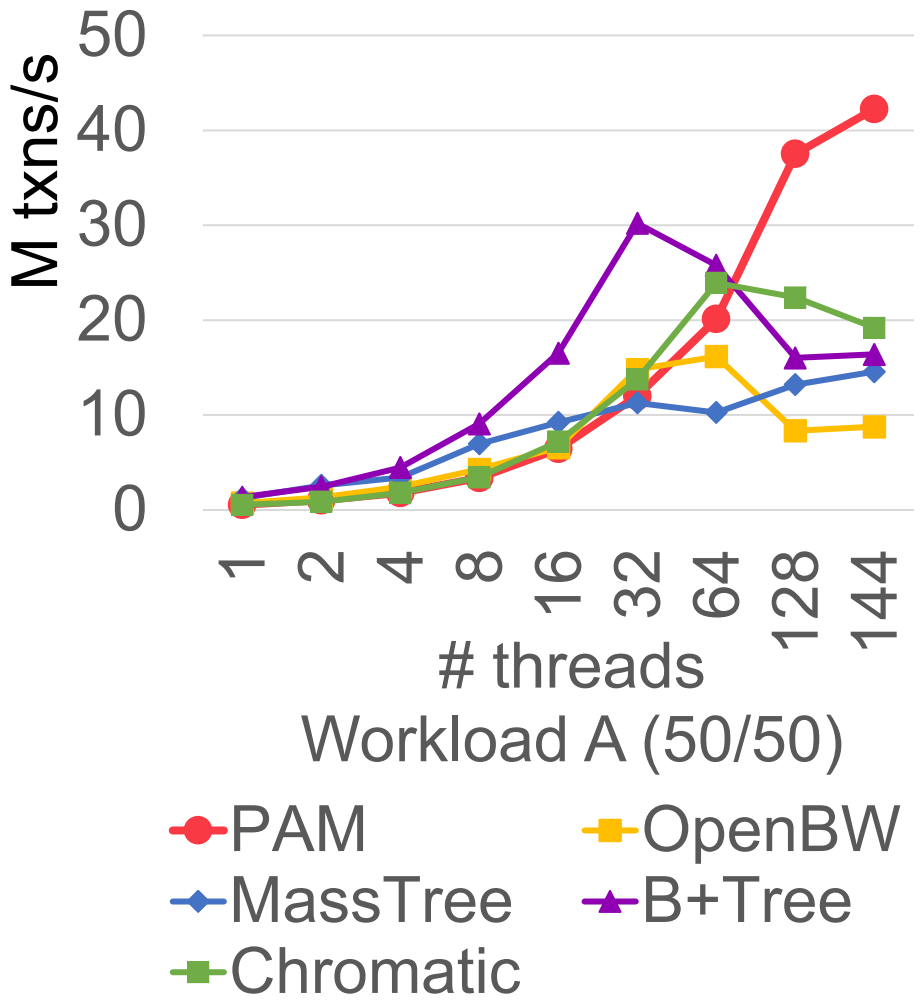


72 cores with hyperthreading, 5×10^7 initial key-value pairs with 10^7 single-operation transactions.

1.2–1.4x Faster than all the other concurrent data structures

Compare to Concurrent Data Structures [SPAA'19]

Throughput: Higher is better



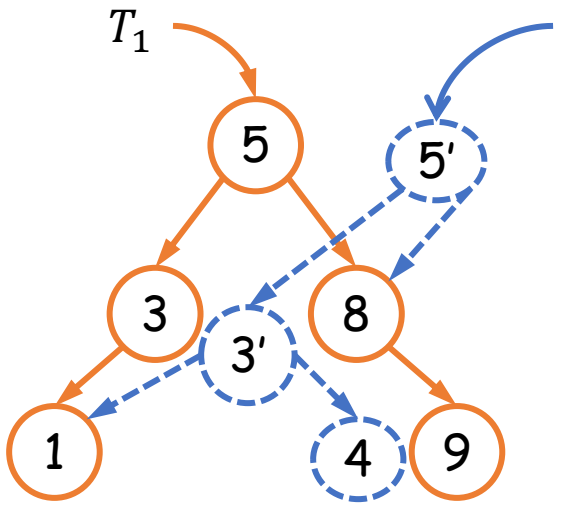
72 cores with hyperthreading, 5×10^7 initial key-value pairs with 10^7 single-operation transactions.

1.2-1.4x Faster than all the other concurrent data structures

(almost-linear) speedup scaling up to **144 Threads**

Garbage Collection

- Reference Counter Garbage Collector
  - Each tree node records the number of other tree nodes/pointers refers to it
  - Node 8 and 1 in the example have reference counter 2

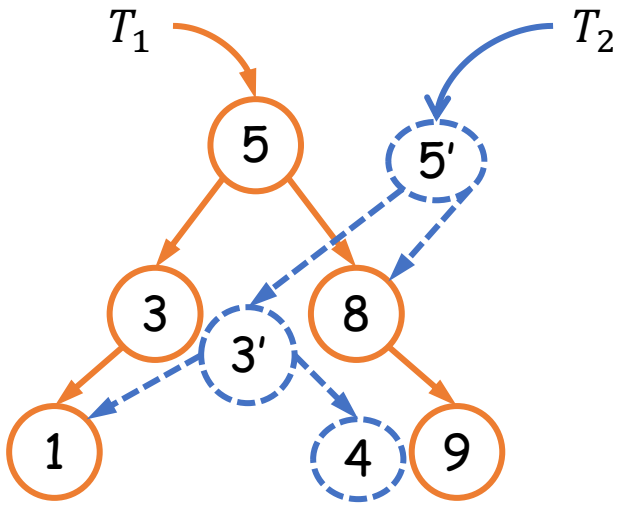


Reference count:

Node	1	3	5	8	9	5′	3′	4
Count	2	1	1	2	1	1	1	1

Garbage Collection [SPAA'19]

- Each tree node records the number of other tree nodes/pointers refers to it
- Node 8 and 1 in the example have reference counter 2
- Collect a node if and only if its reference count is 1




```
collect(node* t) {
  if (!t) return;
  if (t->ref_cnt == 1) {
    node* lc = t->lc, *rc = t->rc;
    free(t);
    in parallel:
      collect(lc);
      collect(rc);
  } else dec(t->ref_cnt);
}
```

Node	1	3	5	8	9	5′	3′	4
Count	1	1	1	1	1	1	1	1

Document Searching

- Inverted index

Document 1:
Blue whales are the largest animals ever known to have lived on Earth.

Document 2:
Elephants are the largest land mammals.

Document 3:
Banging your head against a wall for one hour burns 150 calories.

Document 4:
Blue whales eat half a million calories in one mouthful.

Word	Document list
…	…
blue	1
whale	1
earth	1
largest	1, 2
calories	3
mammals	2
head	3

Searching queries:
Find corresponding document lists

Add a new document:
All updates need to be done atomically

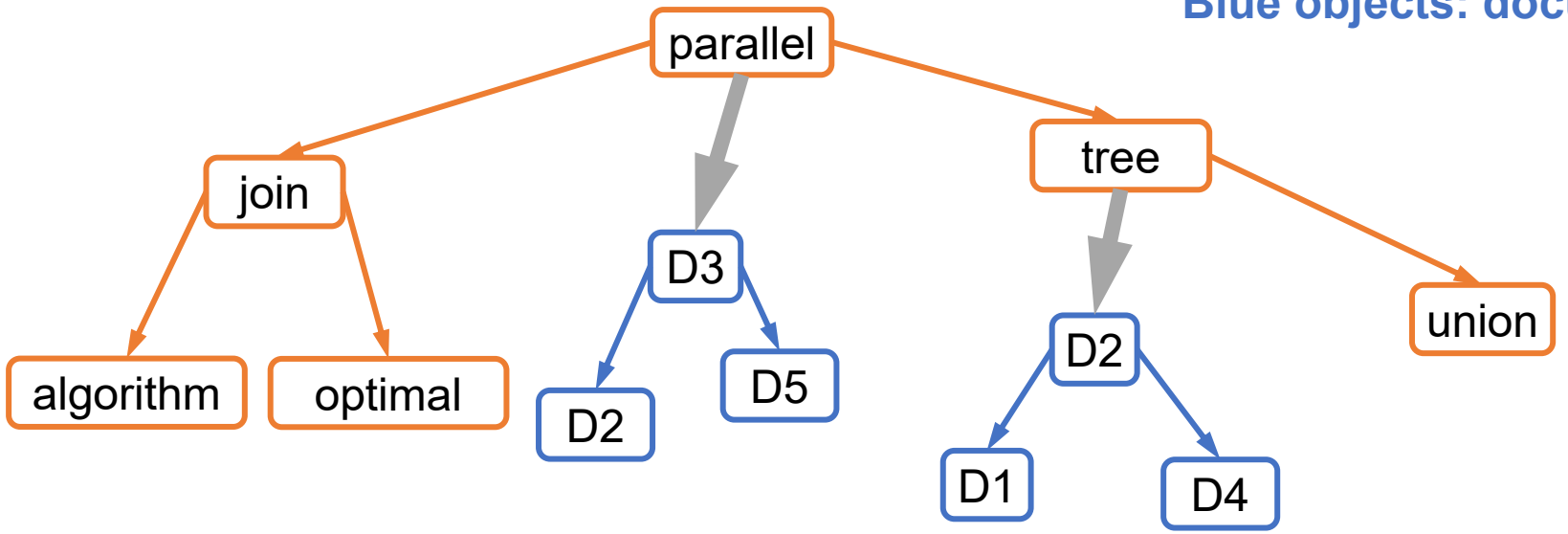
Add document 4

Add “4” to the lists of {Blue, whales, eat, half, million, calories, mouthful}

Inverted Indexes

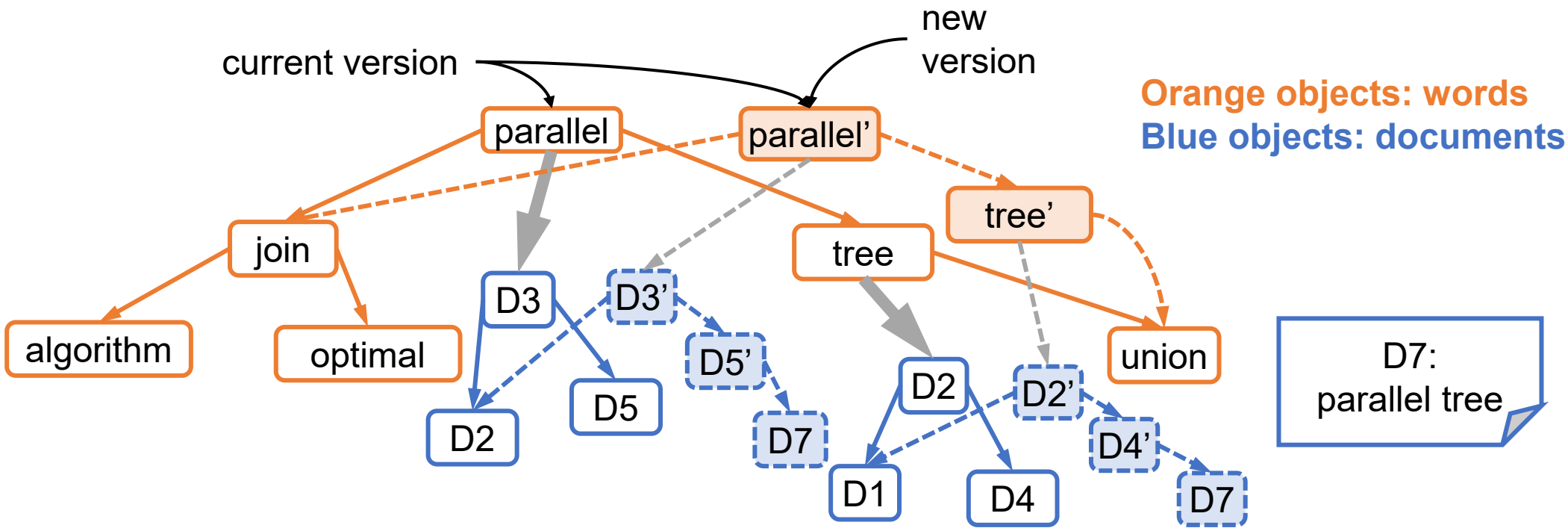
- Stores each word in the outer tree with its document list as an inner tree
- Queries: concurrent analytical queries are done on the current version
  - OR/AND query: union/intersection on the two document lists
  - Nested trees: represent the affiliation relation

Orange objects: words
Blue objects: documents



Inverted Indexes

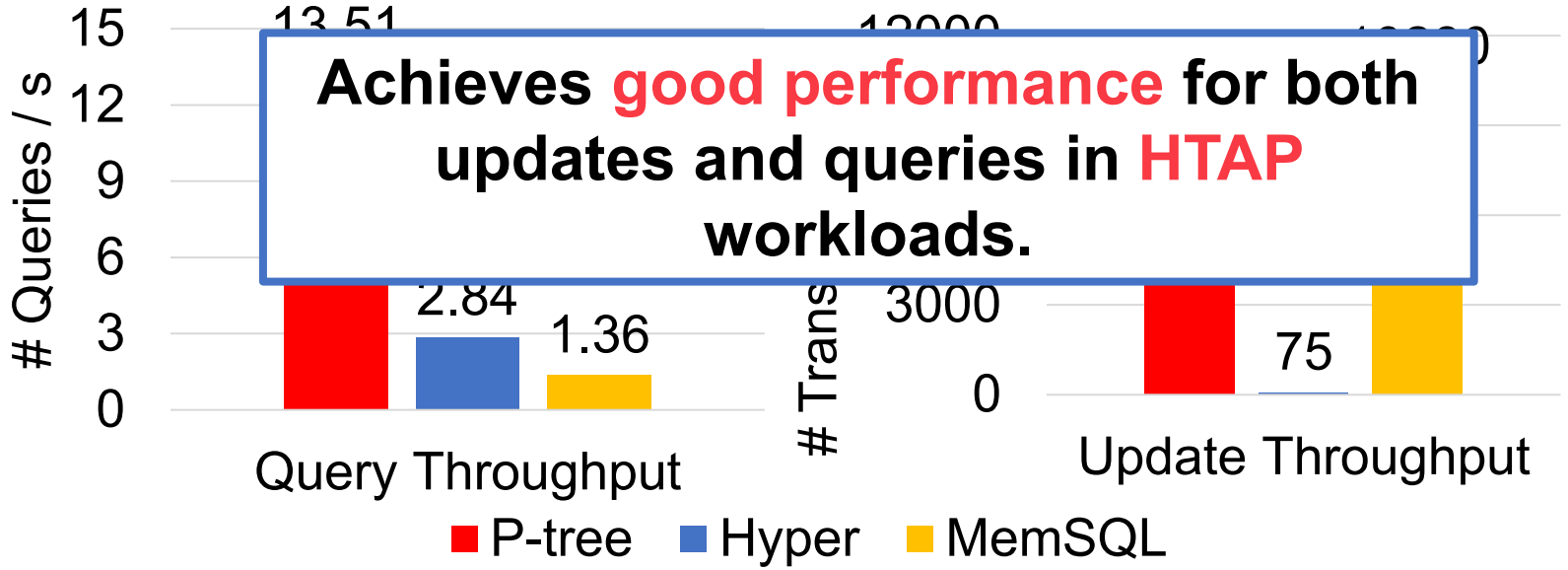
- Updates: take a Union with path-copying (update inner tree for duplicates)
  - Atomic: make multiple word-doc pairs visible at once
  - Non-destructive: old versions for ongoing queries




```
new_version = current_version.add((parallel, D7), (tree, D7))
current_version = new_version
```

Experiments on TPC Benchmarks

- 100 GB data, 72 cores with hyperthreading
- Compare to HyPer [Neumann et al.] and MemSQL [Shamgnov'14]



4-9x

faster than both systems in queries

Comparable

performance to the best of the previous in updates

All throughput numbers: **Higher is better**

Applications using join-based MVCC

- Inverted index searching
- Indexes for HTAP database systems
- Transactional systems with precise GC
- Graph processing

- Example code for all applications available on Github
- Algorithms and more experiments available:
  - (library) PAM: Parallel Augmented Maps, Yihan Sun, Daniel Ferizovic and Guy Blelloch, PPoPP 2018
  - (version control, garbage collection) Multiversion Concurrency with Bounded Delay and Precise Garbage Collection, Naama Ben-David, Guy E. Blelloch, Yihan Sun and Yuanhao Wei, SPAA 2019.
  - (graph processing, compression) Low-Latency Graph Streaming Using Compressed Purely-Functional Trees, Laxman Dhulipala, Guy Blelloch, and Julian Shun, PLDI 2019
  - (database system) On Supporting Efficient Snapshot Isolation for Hybrid Workloads with Multi-Versioned Indexes, Yihan Sun, Guy E. Blelloch, Wan Shen Lim and Andrew Pavlo, PVLDB, 13(2).

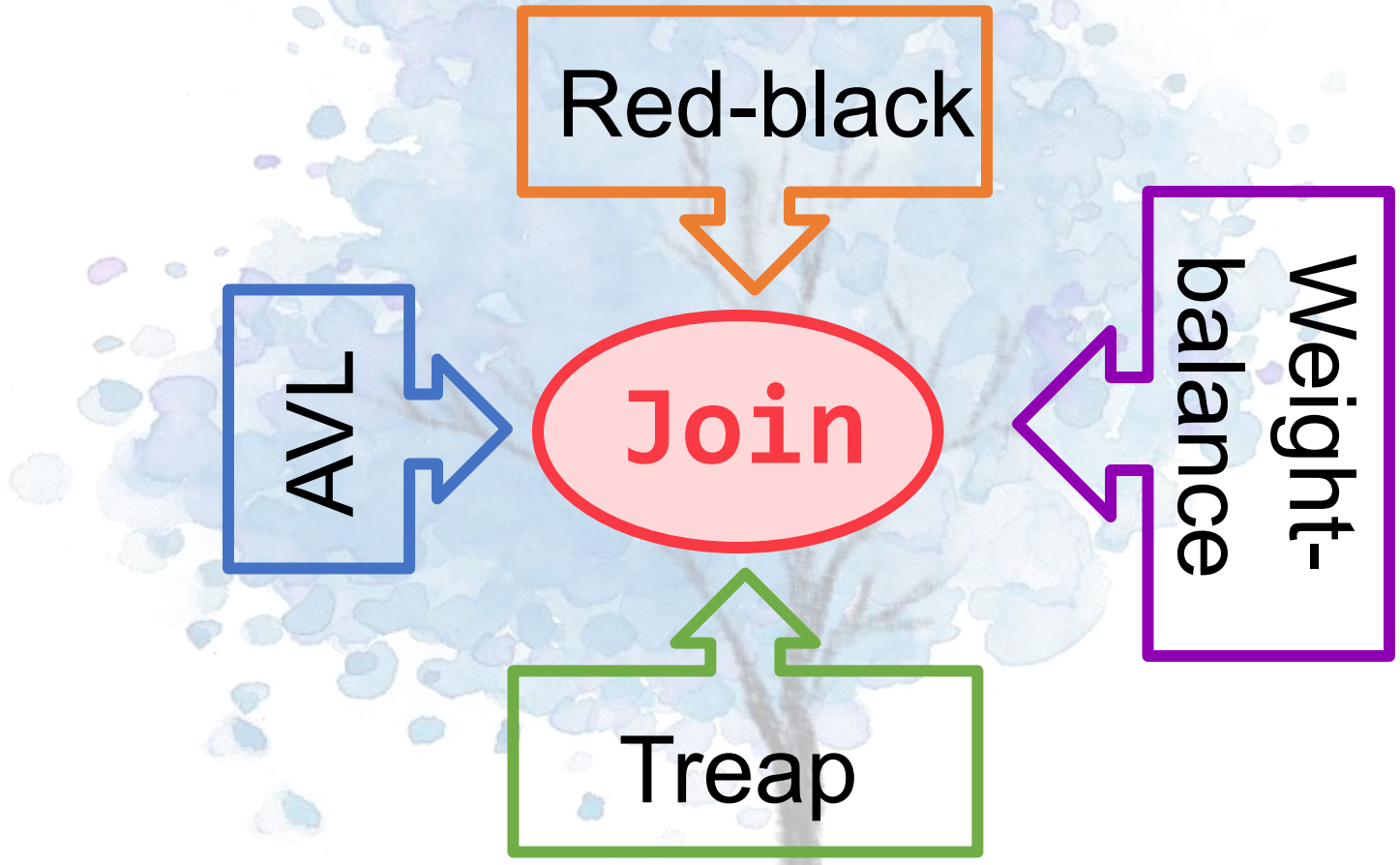
Outline

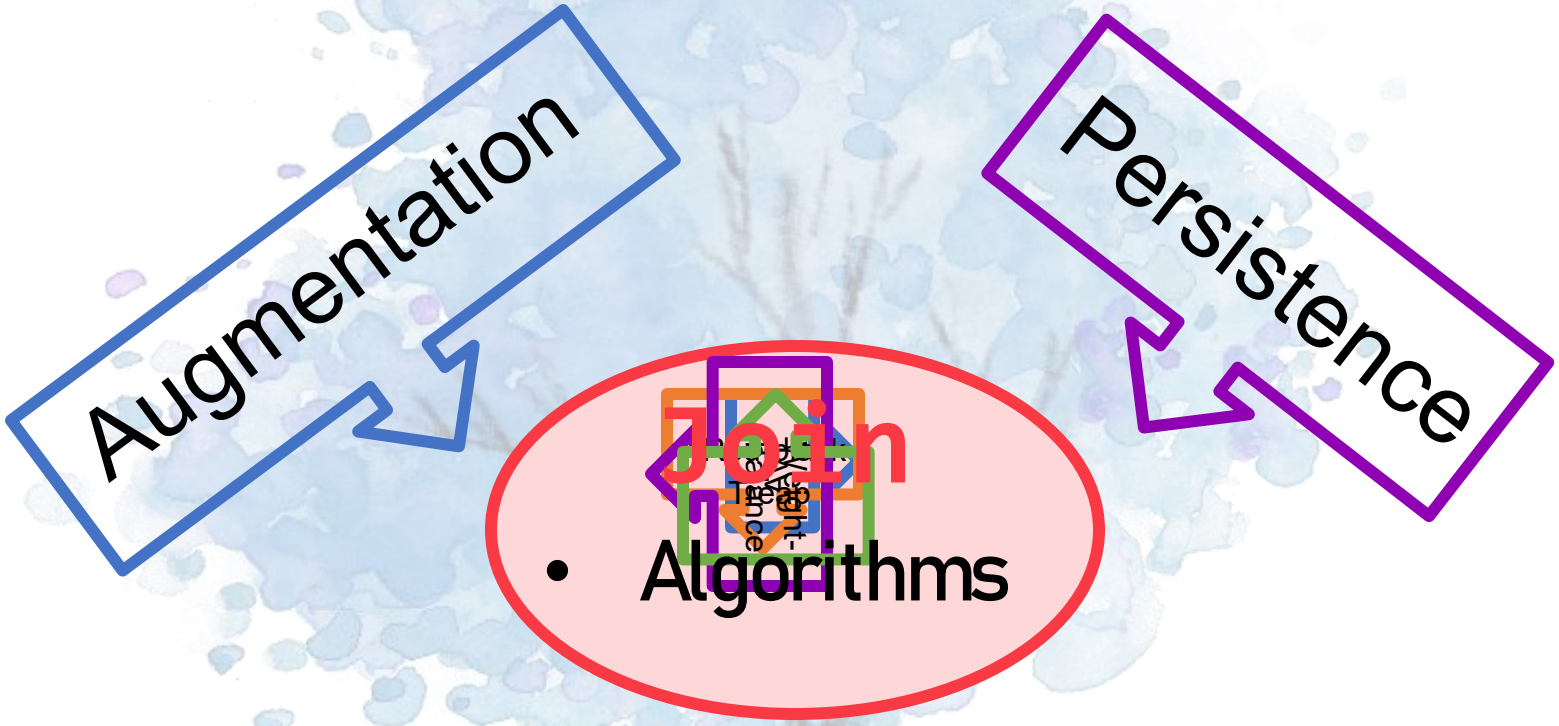
1 Algorithms Using Join

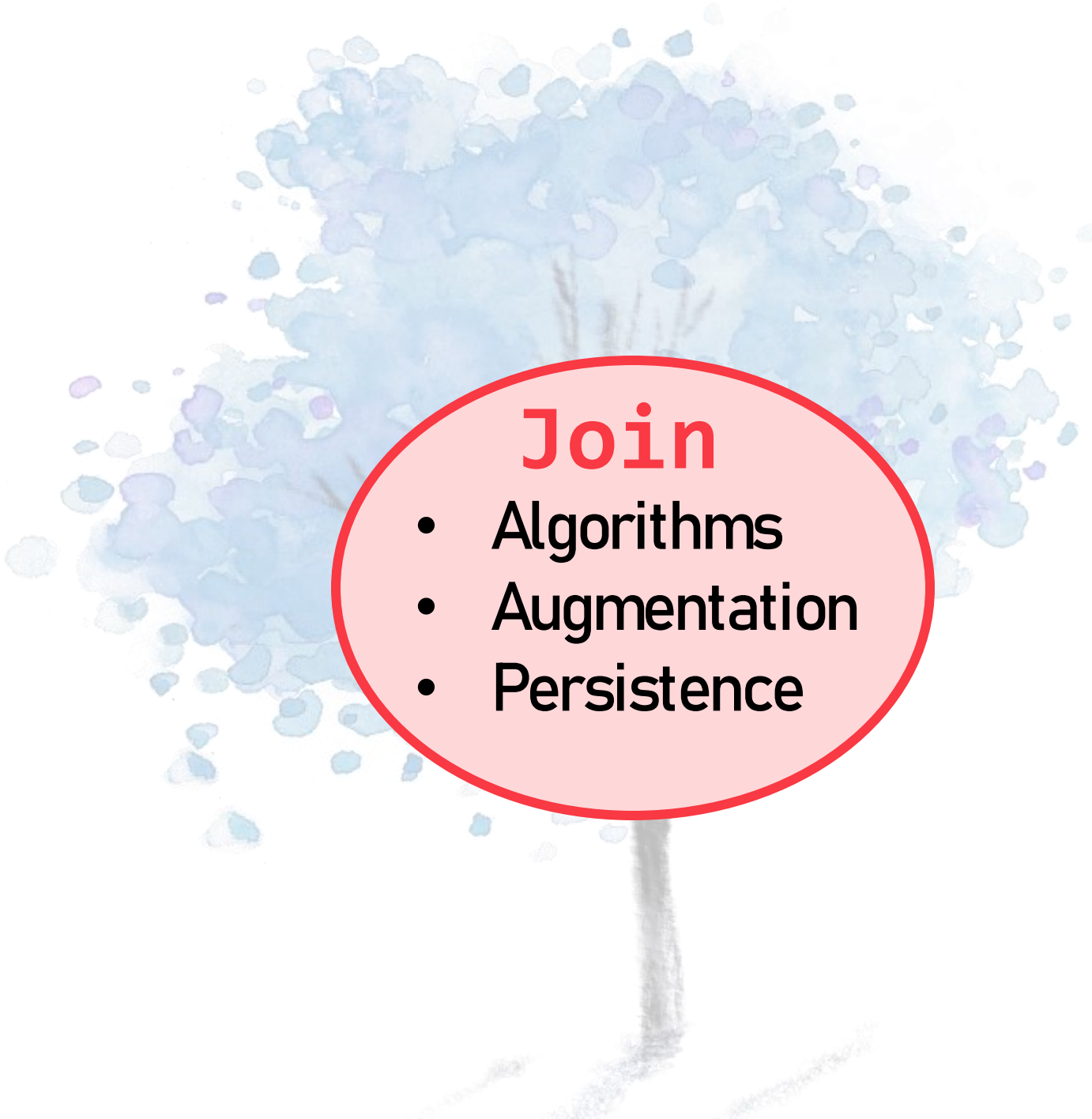
2 Augmentation Using Join

3 **Persistence Using Join**

- Multi-version concurrency control (MVCC)
- Database systems with concurrent updates and queries
- Fast in practice


Red-black
AVL
Join
Weight-balance
Treap


Augmentation
Persistence
Join
• Algorithms



Join

- Algorithms
- Augmentation
- Persistence

In theory:
Join-based algorithms

Join

- Algorithms
- Augmentation
- Persistence

In practice:
The PAM library

- Simple
- Supporting a wide range of algorithms
- Work-optimal and poly-log span
- Balancing scheme independent
- Abstract augmentation
- Persistent & multi-versioning

P-trees

Concise implementation

High performance both sequentially and in parallel

Interval Trees

2D range query
2D segment query
2D rectangle query

Inverted Index
Searching

Database Management Systems