CSE 6230:
HPC Tools and Applications

# Lecture 18: Graph Representations

**Helen Xu**
**hxu615@gatech.edu**

# Recap: Graph representations

Vertices labeled from 0 to n-1

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 1 | 0 | 0 | 0 |

Adjacency matrix
("1" if edge exists,
"0" otherwise)

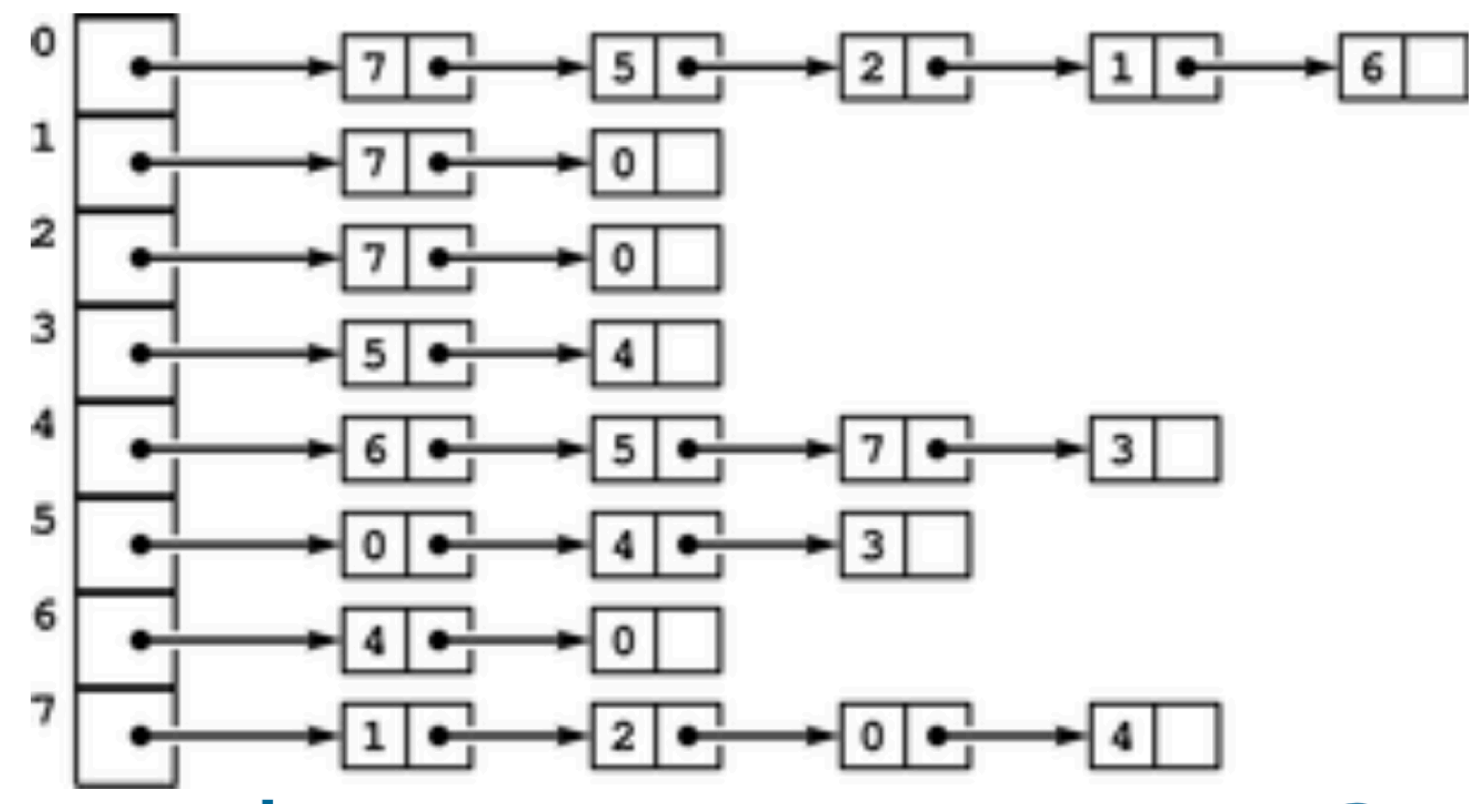
(0,1)
(1,0)
(1,3)
(1,4)
(2,3)
(3,1)
(3,2)
(4,1)

Edge list

# Recap: Graph representations

Adjacency list

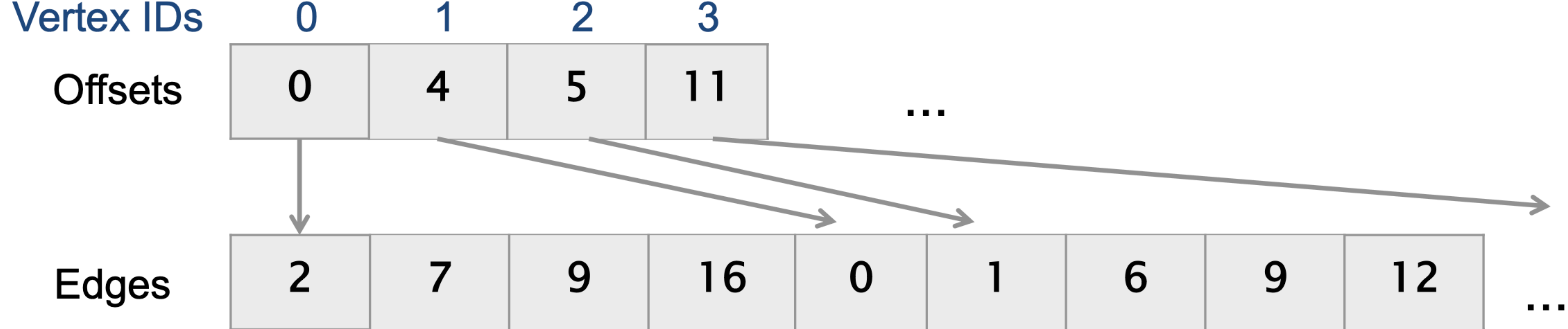
- Array of pointers (one per vertex)
- Each vertex has an unordered list of its edges
- Can substitute linked lists with arrays for better cache performance (at the cost of updatability)



# Recap: Graph representations

Compressed sparse row (CSR)

- Two arrays: Offsets and Edges
- Offsets[i] stores the offset of where vertex i's edges start in Edges



# Tradeoffs in graph representations

Let m = number of edges and n = number of vertices.

| | Adjacency matrix | Edge list **(COO)** | Adjacency list | Compressed sparse row |
|---|---|---|---|---|
| Storage cost / scanning whole graph | | | | |
| Add edge | | | | |
| Delete edge from vertex v | | | | |
| Finding all neighbors of a vertex v | | | | |
| Finding if w is a neighbor of v | | | | |

**What is the space usage of the different representations?**

From MIT 6.172

# Tradeoffs in graph representations

Let m = number of edges and n = number of vertices.

| | Adjacency matrix | Edge list **(COO)** | Adjacency list | Compressed sparse row |
|---|---|---|---|---|
| Storage cost / scanning whole graph | $O(n^2)$ | $O(m)$ | $O(m+n)$ | $O(m+n)$ |
| Add edge | | | | |
| Delete edge from vertex v | | | | |
| Finding all neighbors of a vertex v | | | | |
| Finding if w is a neighbor of v | | | | |

**What is the cost to add and delete edges in different representations?**

# Tradeoffs in graph representations

Let m = number of edges and n = number of vertices.

| | Adjacency matrix | Edge list **(COO)** | Adjacency list | Compressed sparse row |
|---|---|---|---|---|
| Storage cost / scanning whole graph | $O(n^2)$ | O(m) | O(m+n) | O(m+n) |
| Add edge | O(1) | O(1) | O(1)/O(deg(v)) | O(m+n) |
| Delete edge from vertex v | O(1) | O(m) | O(deg(v)) | O(m+n) |
| Finding all neighbors of a vertex v | | | | |
| Finding if w is a neighbor of v | | | | |

**What is the cost to find all neighbors of a given vertex in different representations?**

# Tradeoffs in graph representations

Let m = number of edges and n = number of vertices.

| | Adjacency matrix | Edge list **(COO)** | Adjacency list | Compressed sparse row |
|---|---|---|---|---|
| Storage cost / scanning whole graph | $O(n^2)$ | O(m) | O(m+n) | O(m+n) |
| Add edge | O(1) | O(1) | O(1)/O(deg(v)) | O(m+n) |
| Delete edge from vertex v | O(1) | O(m) | O(deg(v)) | O(m+n) |
| Finding all neighbors of a vertex v | O(n) | O(m) | O(deg(v)) | O(deg(v)) |
| Finding if w is a neighbor of v | **What is the cost to find if w is a neighbor of v in different representations?** | | | |

From MIT 6.172

# Tradeoffs in graph representations

Let m = number of edges and n = number of vertices.

| | Adjacency matrix | Edge list **(COO)** | Adjacency list | Compressed sparse row |
|---|---|---|---|---|
| Storage cost / scanning whole graph | $O(n^2)$ | O(m) | O(m+n) | O(m+n) |
| Add edge | O(1) | O(1) | O(1)/O(deg(v)) | O(m+n) |
| Delete edge from vertex v | O(1) | O(m) | O(deg(v)) | O(m+n) |
| Finding all neighbors of a vertex v | O(n) | O(m) | O(deg(v)) | O(deg(v)) |
| Finding if w is a neighbor of v | O(1) | O(m) | O(deg(v)) | O(deg(v)) |

# Tradeoffs in graph representations

Let m = number of edges and n = number of vertices.

| | Adjacency matrix | Edge list **(COO)** | Adjacency list | Compressed sparse row |
|---|---|---|---|---|
| Storage cost / scanning whole graph | $O(n^2)$ | O(m) | O(m+n) | O(m+n) |
| Add edge | O(1) 💚 | O(1) | O(1)/O(deg(v)) | O(m+n) 😡 |
| Delete edge from vertex v | O(1) | O(m) | O(deg(v)) | O(m+n) |
| Finding all neighbors of a vertex v | O(n) 😡 | O(m) | O(deg(v)) | O(deg(v)) 💚 |
| Finding if w is a neighbor of v | O(1) | O(m) | O(deg(v)) | O(deg(v)) |

**Tradeoff**

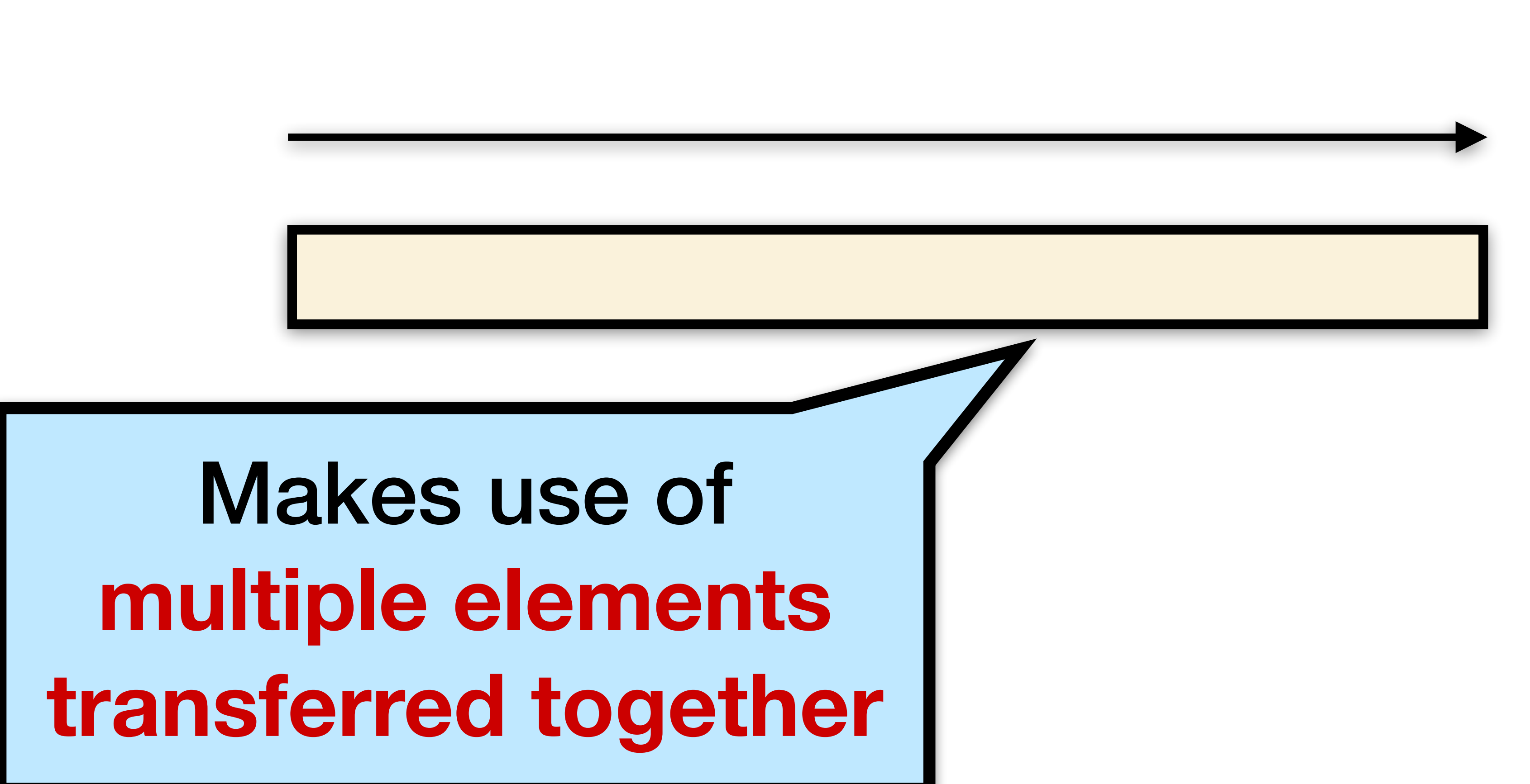
**Tradeoff**

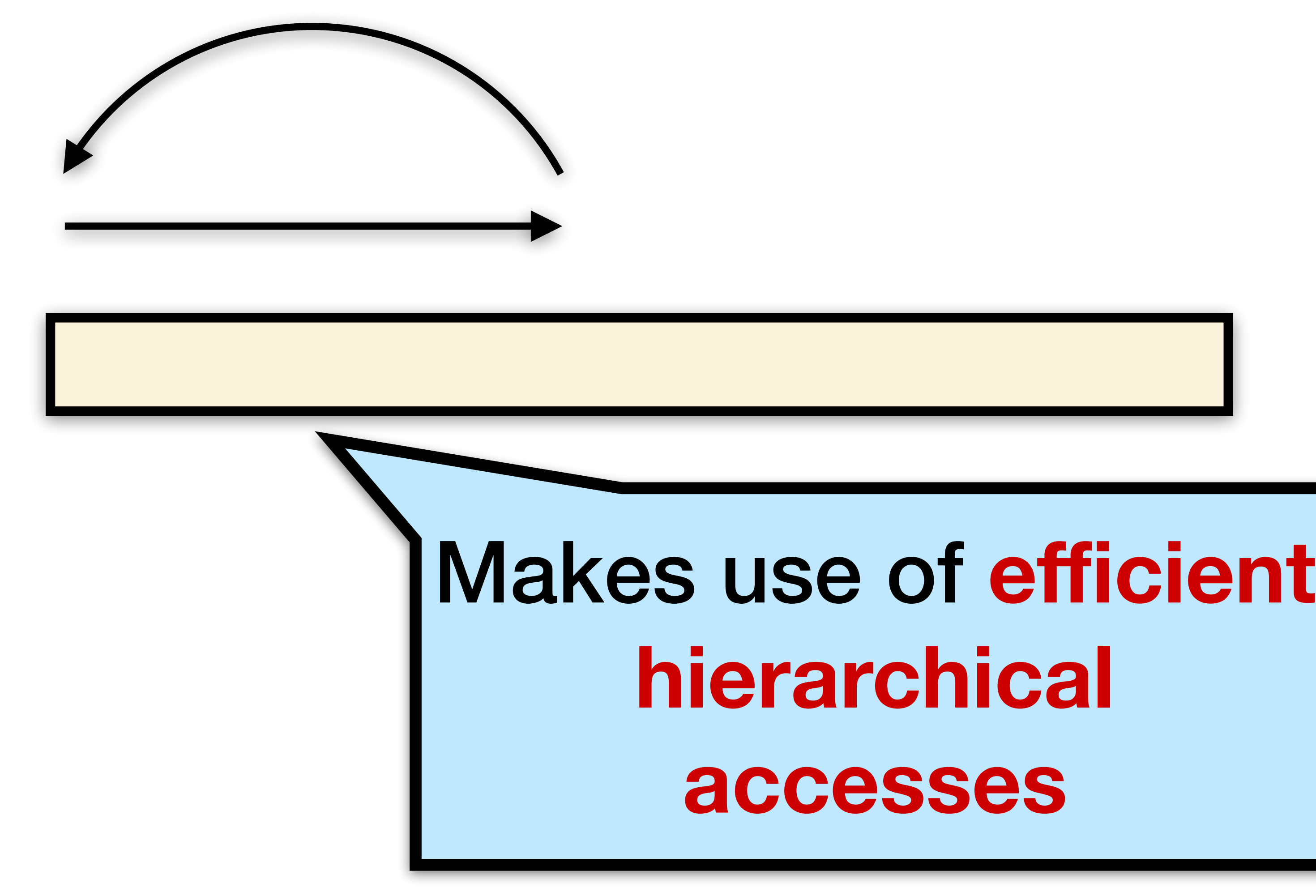
From MIT 6.172

# Recall: Spatial and Temporal Locality

**Spatial locality:** how many accesses an algorithm makes to **nearby** data over a short period of time [Denning72, Denning05].

**Temporal locality:** how many repeated accesses an algorithm makes to **the same** data over a short period of time [Denning72, Denning05].




**Question:** Which type should we be targeting in general for graph optimization?

# Spatial Locality Determines Graph Query Performance

Dynamic-graph data structures must support fast graph queries.

**Vertex scans**, or the processing of a vertex's incident edges, are a crucial step in many graph queries [ShunBl13].

```
Input: graph G, source vertex src
let Q be a queue
label src as explored
Q.enqueue(src)
while Q is not empty:
  v = Q.dequeue()
  for all edges (v, w) in G.neighbors(v):
    if w not explored:
      label w as explored
      Q.enqueue(w)
```

Scan

Breadth-first search

```
Input: graph G
let triangle_count = 0
let E = G.edges()
for (u, v) in E:
  intersect neighbors of u and v:
    if u and v share a neighbor w:
      triangle_count++;
```

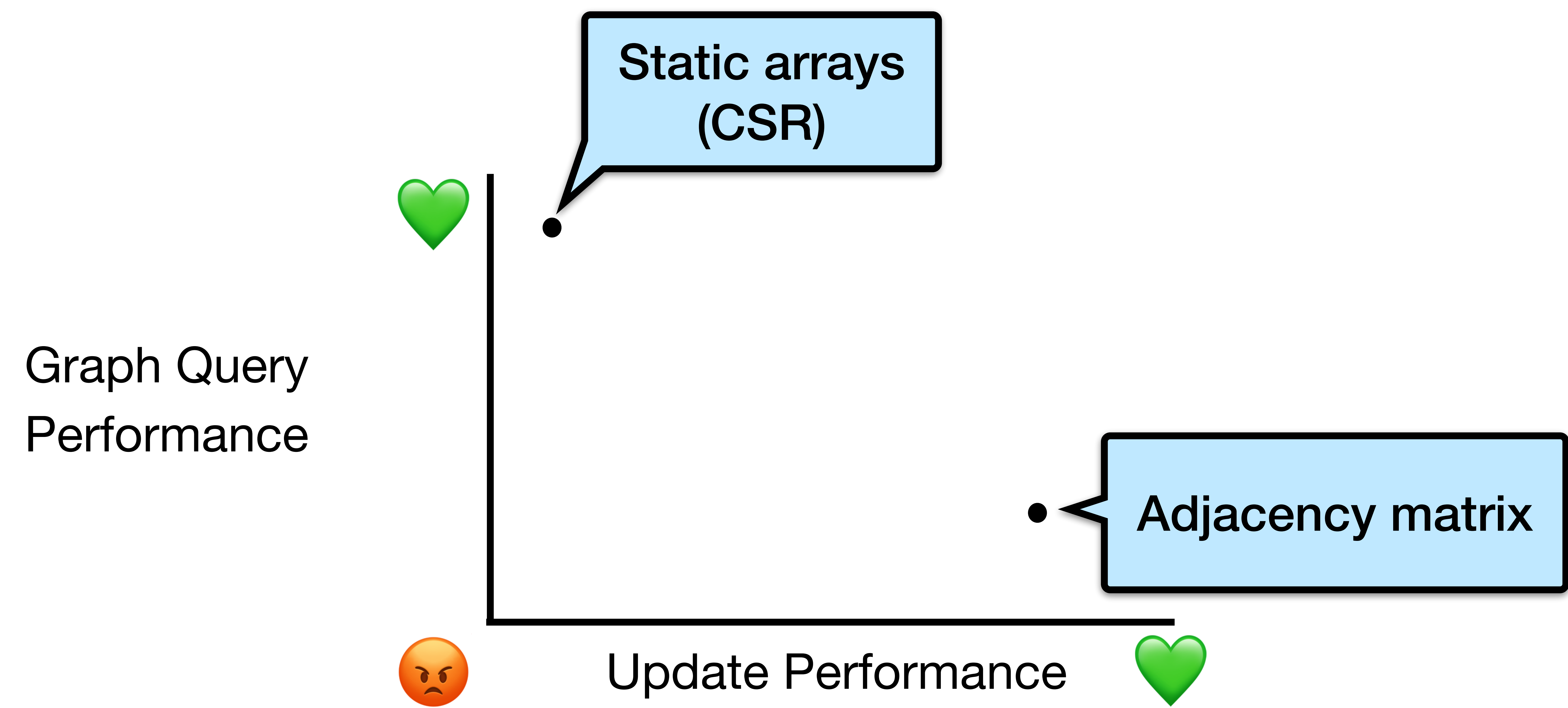
Scan

Triangle counting

Each neighbor list is scanned at most once (no temporal locality), so optimize for **spatial locality**

# Tradeoff between Locality and Updatability

**Problem: Can we choose data structures to support efficient scans and updates?**

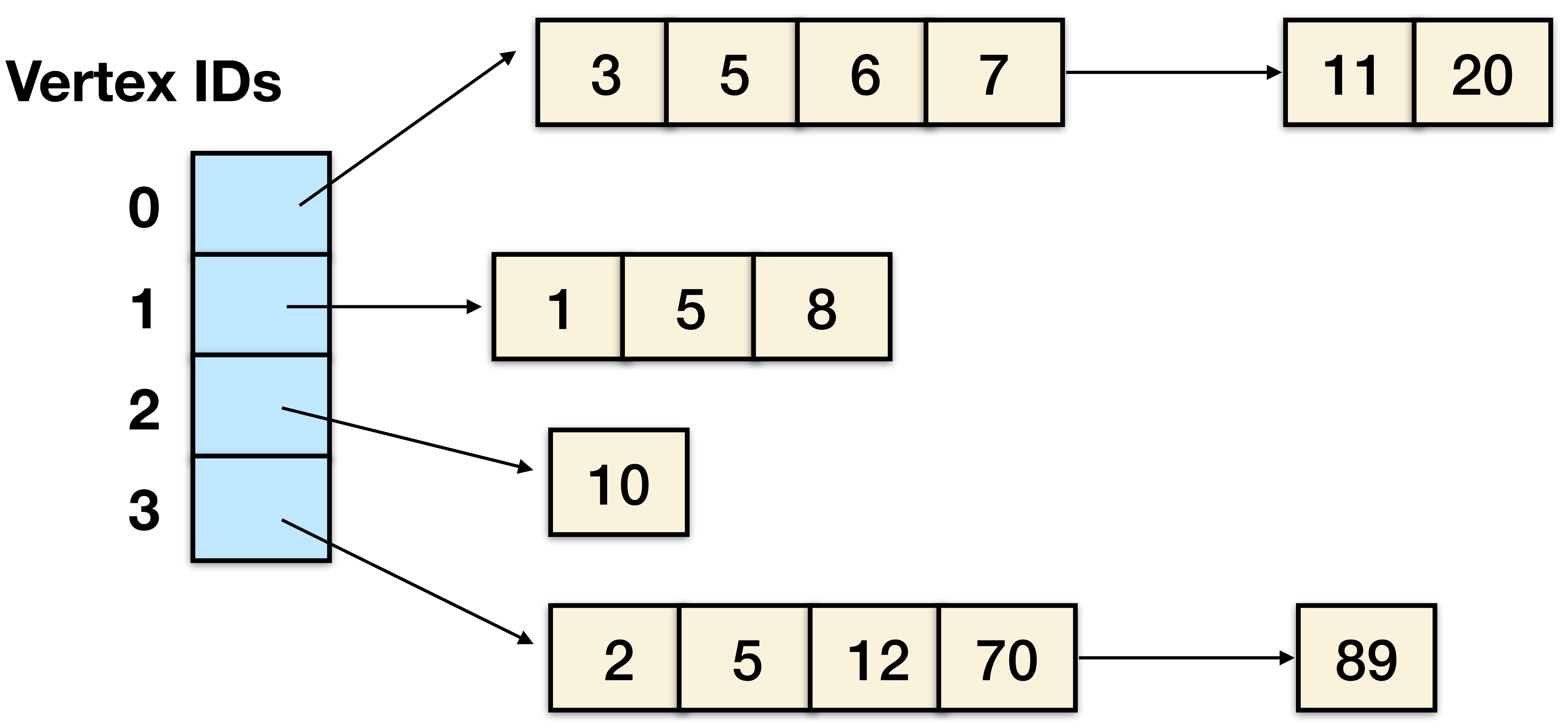


# STINGER: High Performance Data Structure for Streaming Graphs
# (Ediger, McColl, Riedy, Bader - HPEC 12)

http://www.stingergraph.com/

# STINGER Data Structure Design

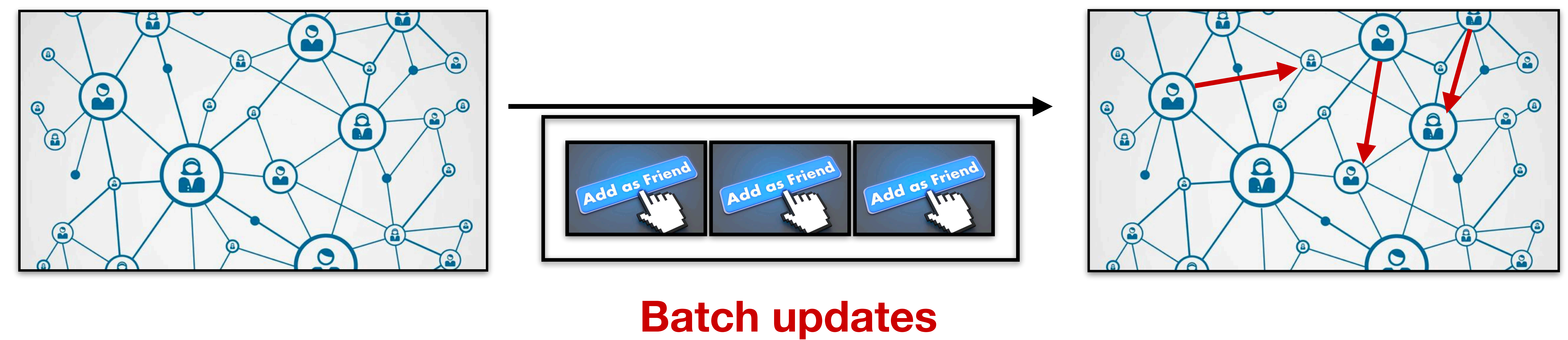
STINGER is based on **linked lists of blocks** - edges incident to a given vertex are stored in a linked list of edge blocks.




Blocked linked lists **improve locality** compared to regular adjacency lists.

# Batch Updates

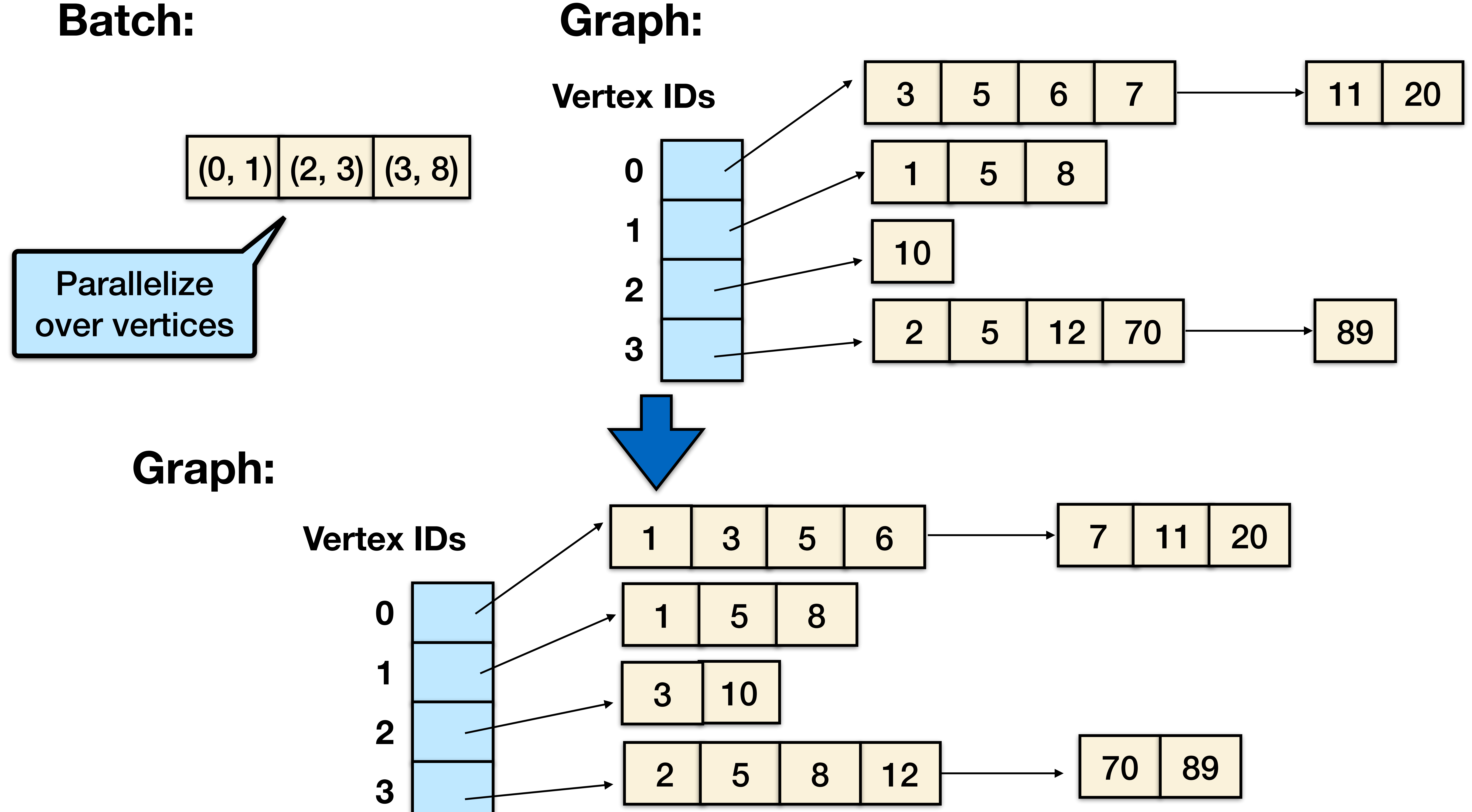
Modern dynamic data structure libraries (including those for graphs) implement **parallel batch updates** which insert/delete many elements at the same time [BarbuzziMiBiBo10, DhulipalaBlSh19, DhulipalaBlGuSu22, ErbKoSa14, FriasSi07, SunFeBl18, TsengDhBl19].




**Batch updates**

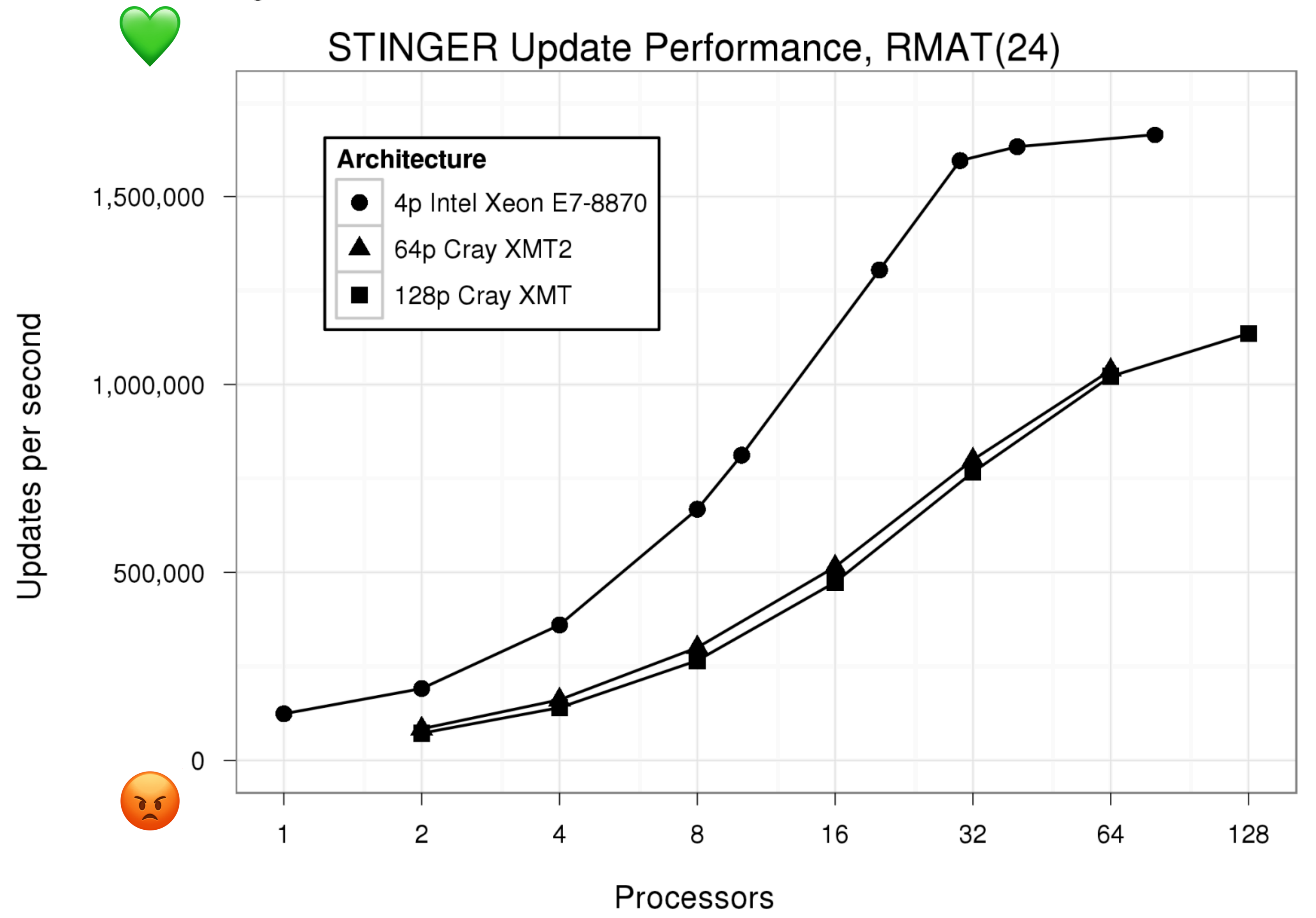
Parallel batch updates **simplify update parallelism and reduce the overall work of each update** by combining multiple updates into one operation on the data structure.

# Batch Updates in STINGER

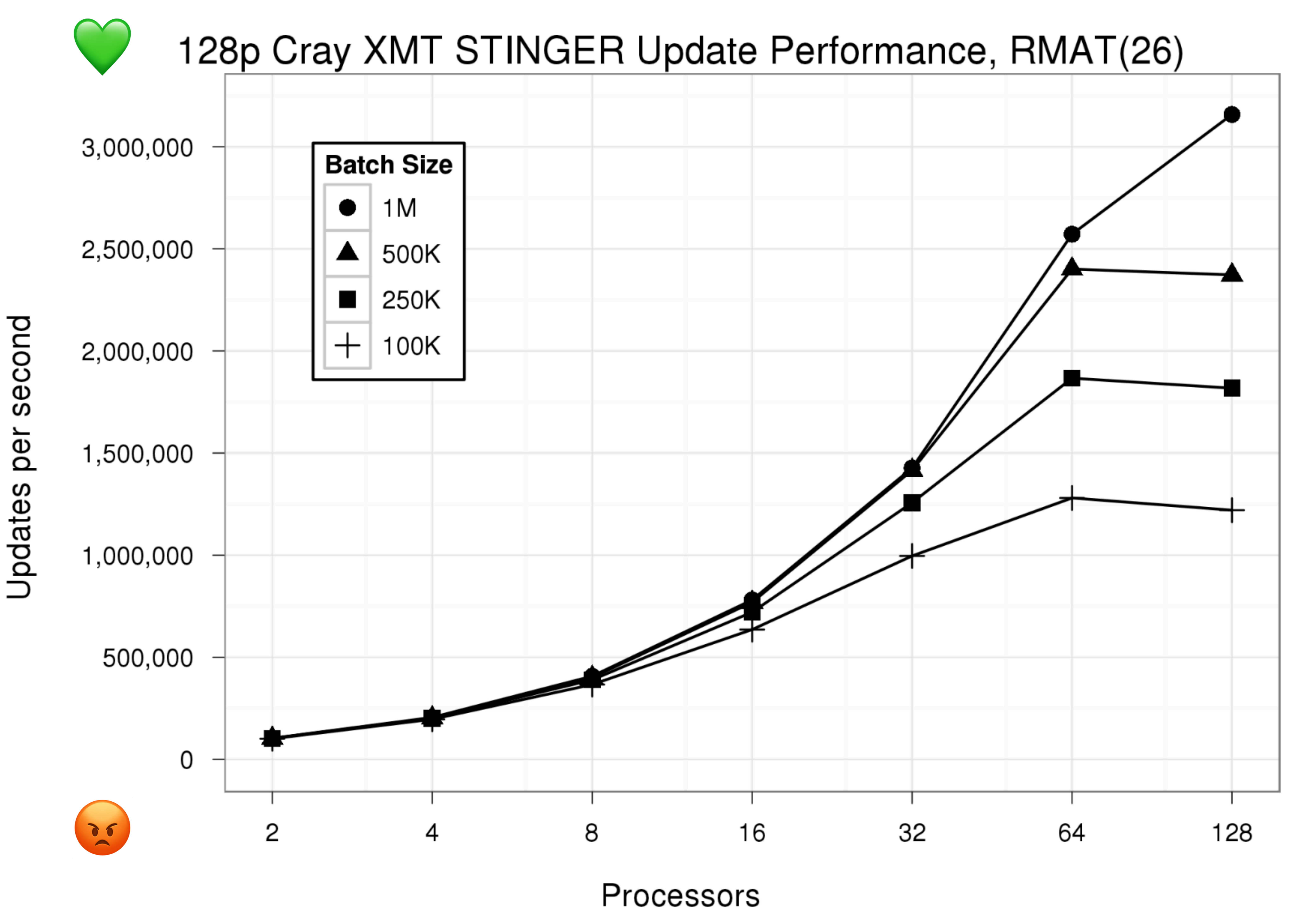


# Update Performance in STINGER

Batch size of 100k edge updates



# Update Performance in STINGER



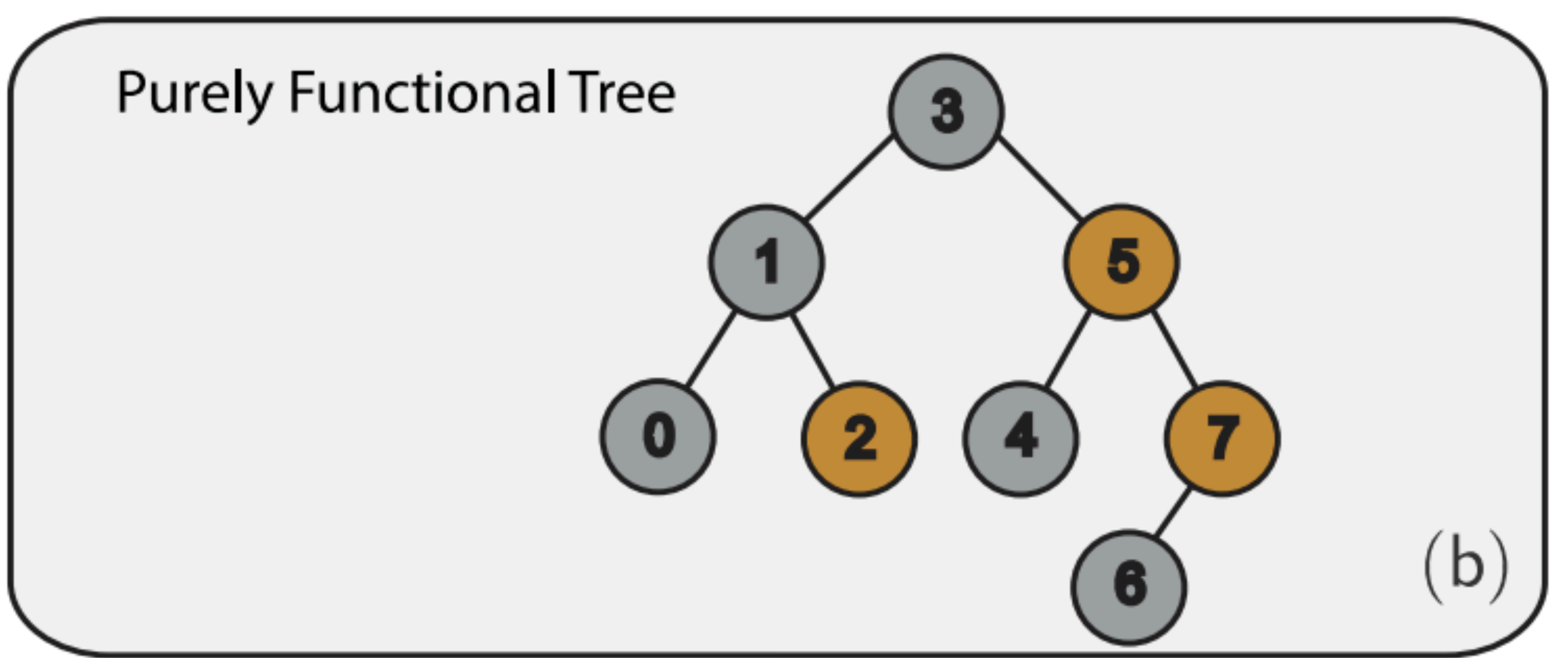
# Low-Latency Graph Streaming using Compressed Purely-Functional Trees
# (Dhulipala, Blelloch, Shun - PLDI 19)

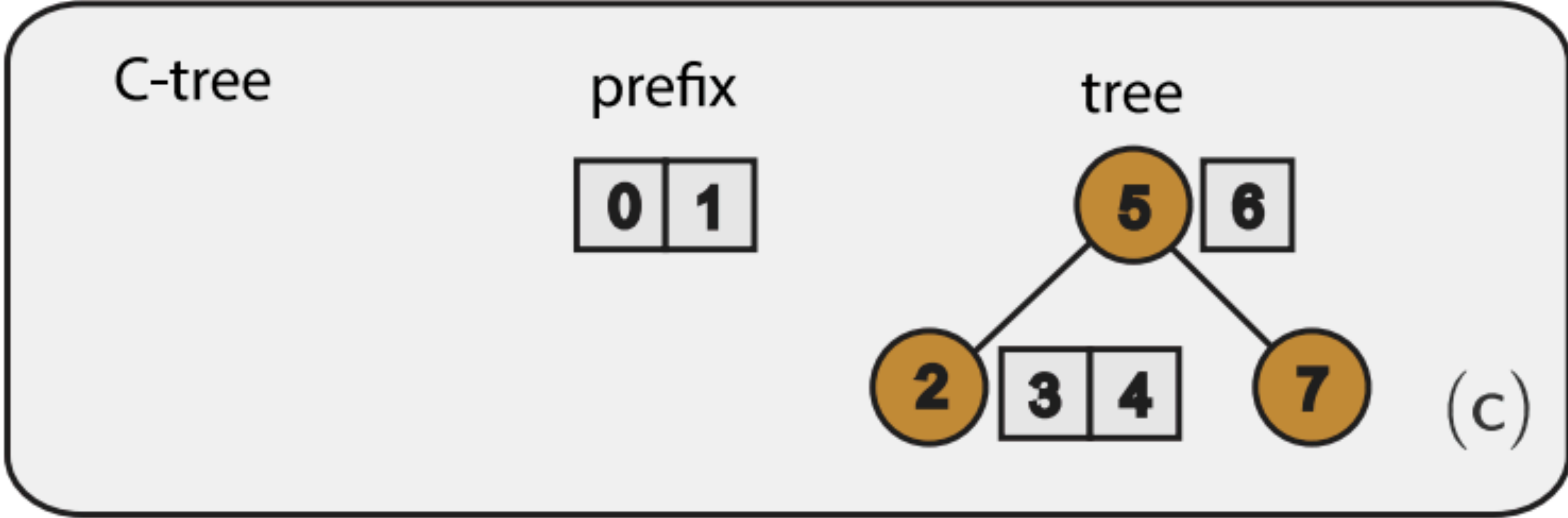
# C-tree Structure

Preserve previous versions of themselves

Compressed **purely-functional** search trees

At a high level, C-trees apply a **chunking scheme** that takes the ordered set of elements to be stored, "promotes" some as **heads** randomly with some probability b, and stores the heads in a tree.



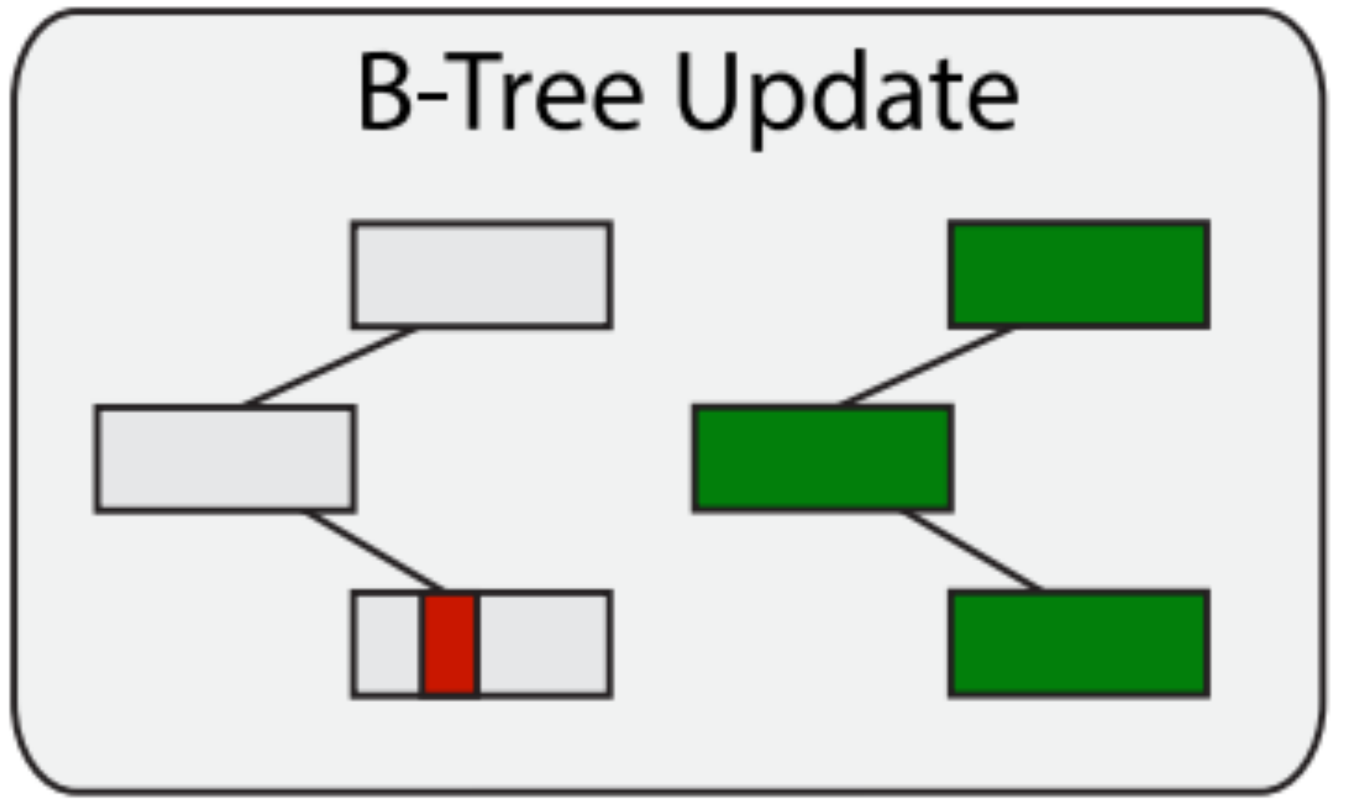

Question: **What is the expected height** of a C-tree in terms of n, the number of elements, and b, the promotion probability?

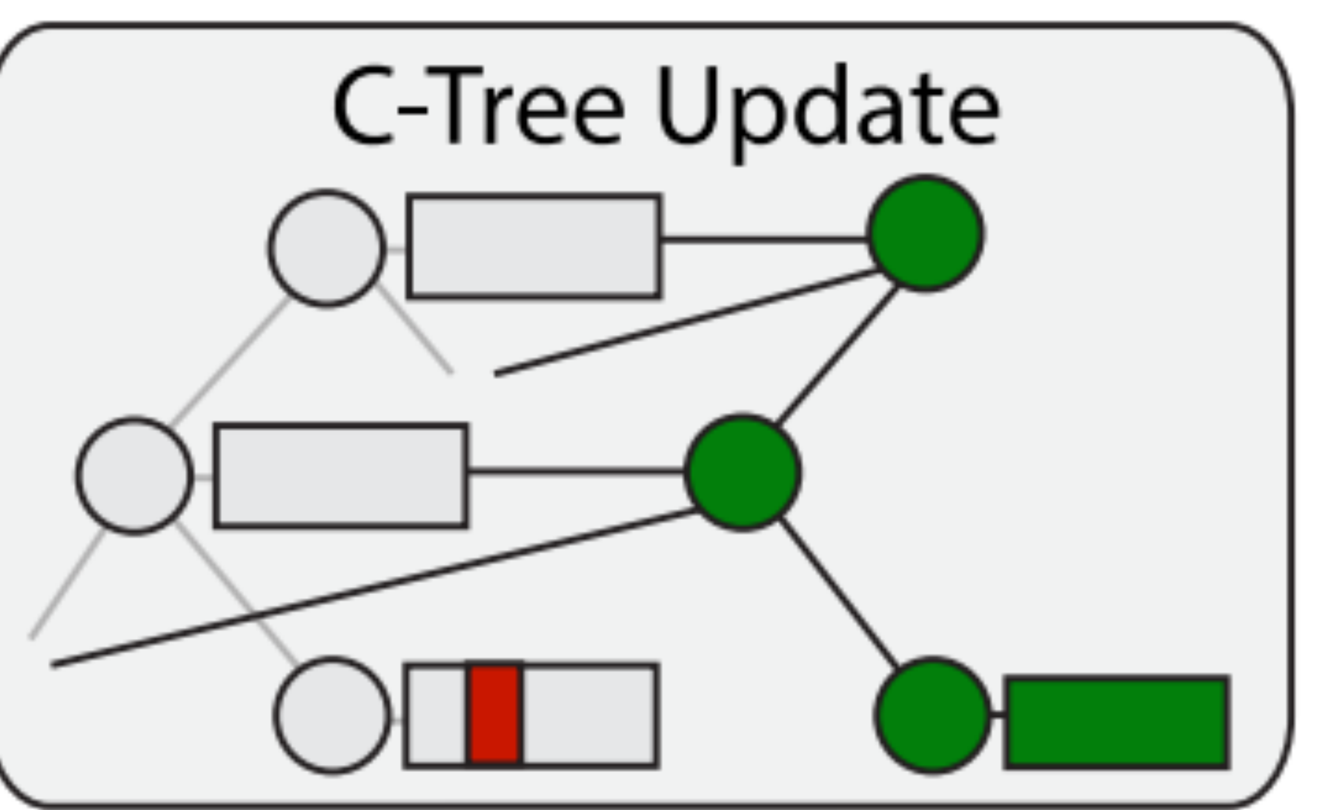
# Updates in C-trees

Why are there C-trees when we already have B-trees?

The problem with B-trees in a purely-functional setting is that the C-tree does **path copying during functional updates**.

- Path copying in B-trees requires copying B pointers per level, while C-trees only need to copy one binary node per level.



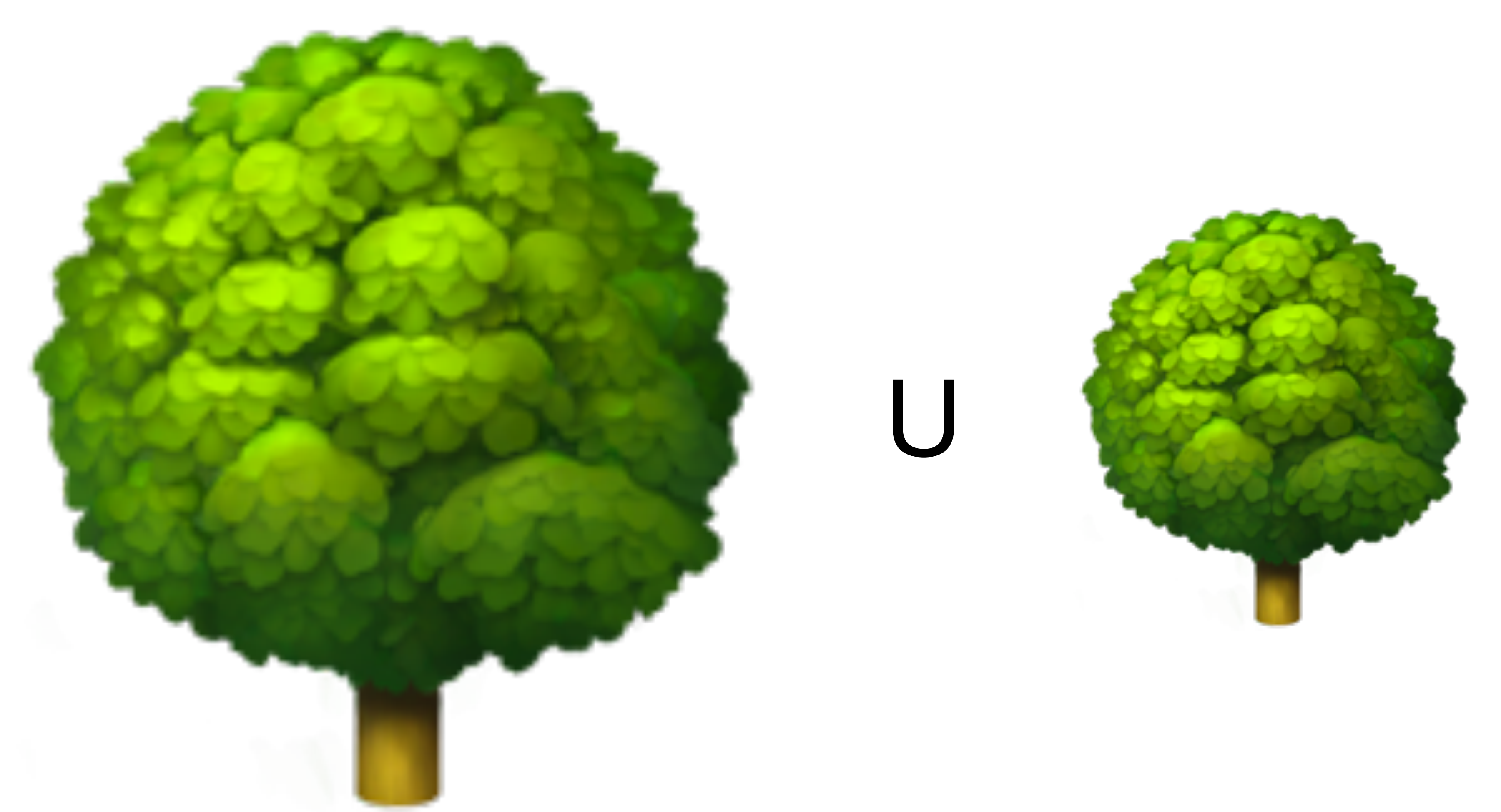



# Batch Updates in C-trees

Batch updates in a C-tree are based on tree union.

Input: Two C-trees, $C_1$ and $C_2$

Output: A C-tree $C$ containing the elements in the union of $C_1$ and $C_2$

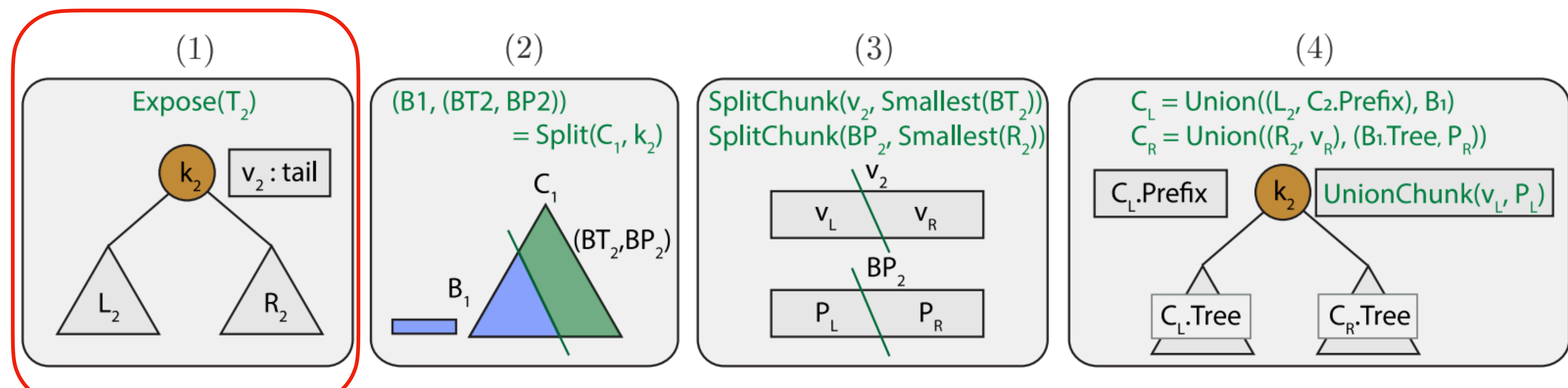


# Batch Updates in C-trees

Step 1 - First call **Expose** on the tree of one of the two C-trees ($C_2$)

Expose input: A C-tree

Expose output: returns the left subtree, element and prefix at the root of the tree, and the right subtree.

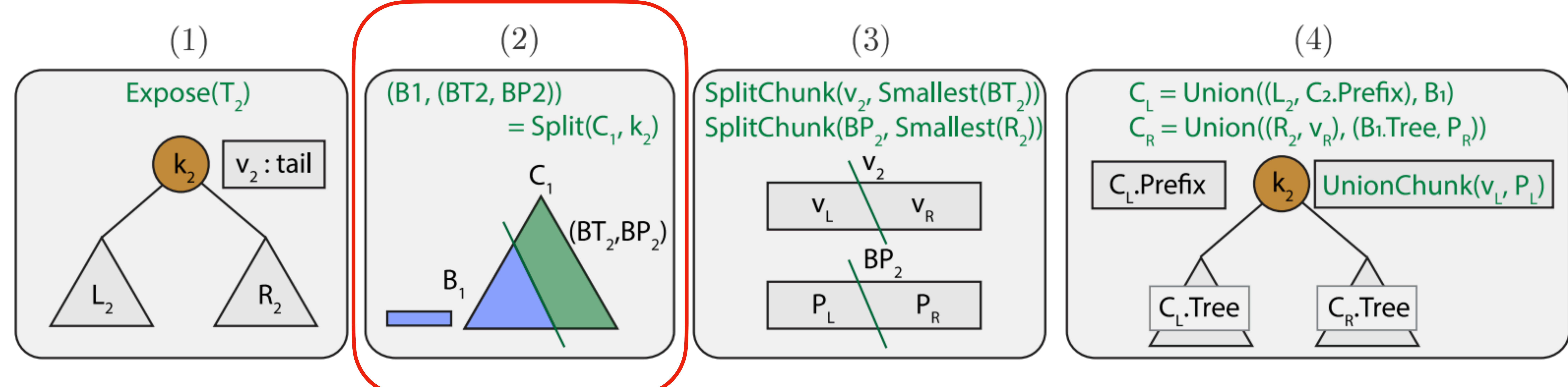


# Batch Updates in C-trees

Step 2 - Using the root of $C_2$, **split** the other tree $C_1$

Split input: A C-tree $B$ and element $k$

Split output: Two C-trees $B_1$ and $B_2$, where $B_1$ (resp. $B_2$) are a C-tree containing all elements less than (resp. greater than) $k$

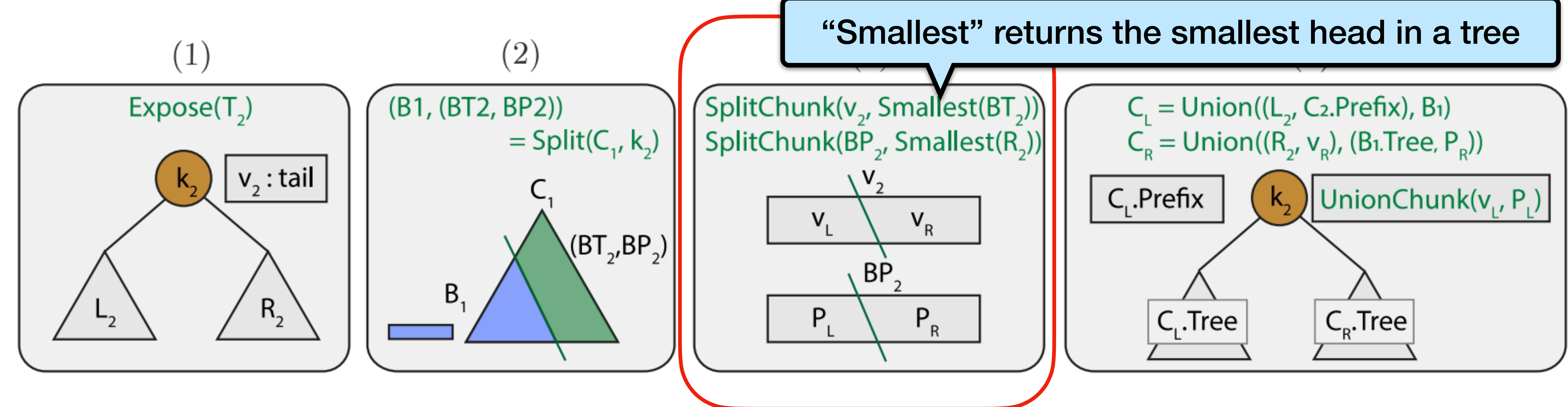


# Batch Updates in C-trees

**Issue:** Some elements in $k_2$'s tail (elements between $k_2$ and the next head) in $C_2$ may come after some heads in $B_2$ ("right" tree of split of $C_1$)

- Similarly, some elements in $B_2$'s prefix may come after some heads of $R_2$ ("right" tree in split of $C_2$).

To handle these, split $v_2$ (tail of $k_2$) by the leftmost element of $B_2$ and split $B_2$'s prefix by the leftmost element of $R_2$.

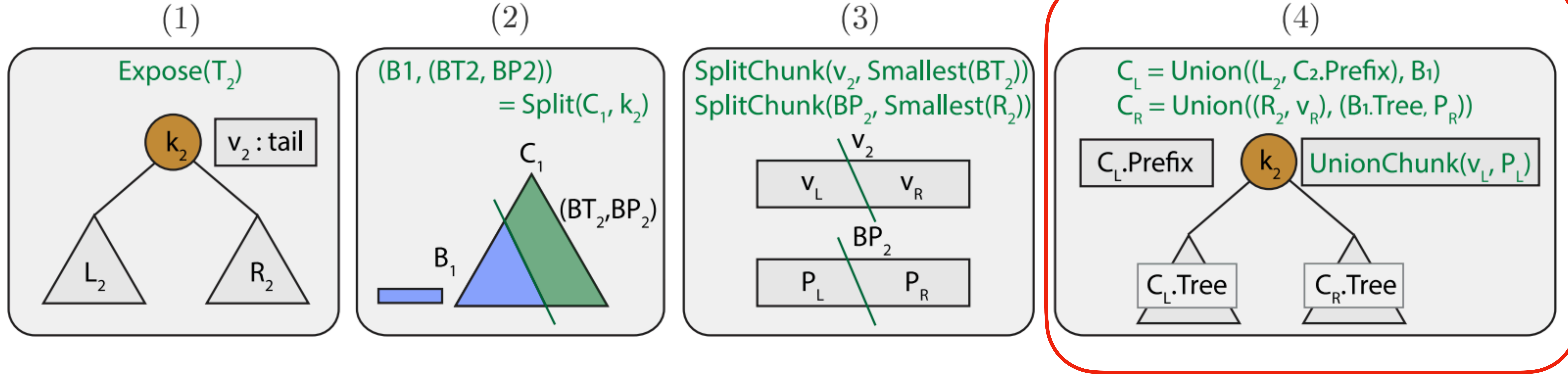


# Batch Updates in C-trees

To proceed, **recursively call Union** on the left and right halves.

- The left recursive call takes as input $B_1$ ("left" half of $C_1$) and $(L_2, P_2)$
- The right recursive call takes the trees $(B_2\,.\,tree, P_R)$ and $(R_2, v_R)$.

The output of Union is a C-tree formed by joining the left and right trees from the recursive calls, $k_2$, and the tail $v_2'$ formed by unioning $v_L$ and $P_L$, with the prefix from $C_L$



# Aspen: Representing Graphs with C-trees

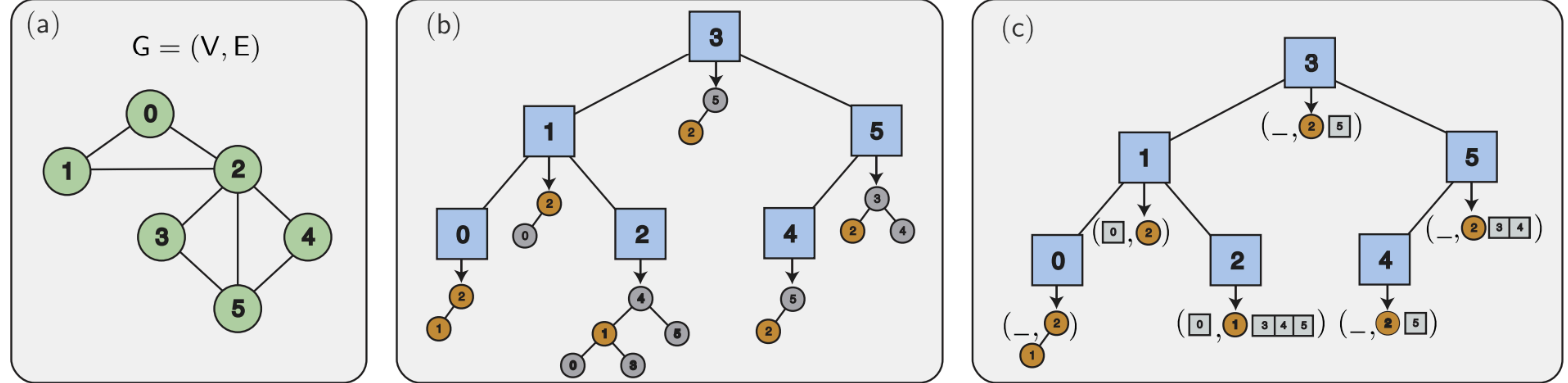



**Figure 4.** We illustrate how the graph (shown in subfigure (a)) is represented as a simple tree of trees (subfigure (b)) and as a tree of *C*-trees (subfigure (c)). As in Figure 1, we color elements (in this case vertex IDs) that are sampled as heads yellow. The prefix and tree in each *C*-tree are drawn as a tuple, following the datatype definition in Figure 1.

# Aspen Batch Update Performance

Batch updates in Aspen use **union** in C-trees on every edge tree.

| Batch Size | Stinger | Updates/sec | Aspen | Updates/sec |
|---|---|---|---|---|
| 10 | 0.0232 | 431 | 9.74e-5 | 102,669 |
| $10^2$ | 0.0262 | 3,816 | 2.49e-4 | 401,606 |
| $10^3$ | 0.0363 | 27,548 | 6.98e-4 | 1.43M |
| $10^4$ | 0.171 | 58,479 | 2.01e-3 | 4.97M |
| $10^5$ | 0.497 | 201,207 | 9.53e-3 | 10.4M |
| $10^6$ | 3.31 | 302,114 | 0.0226 | 44.2M |
| $2 \cdot 10^6$ | 6.27 | 318,979 | 0.0279 | 71.6M |

**Table 8.** Running times and update rates (directed edges/second) for Stinger and Aspen when performing batch edge updates on an empty graph with varying batch sizes. Inserted edges are sampled from the RMAT graph generator. All times are on 72 cores with hyper-threading.

# Aspen Algorithm Performance

Breadth-first search

Betweenness centrality

| App. | Graph | ST | LL | A | A(1) | $A^\dagger$ | ST/A | LL/A |
|---|---|---|---|---|---|---|---|---|
| BFS | LiveJournal | 0.478 | 0.161 | 0.047 | – | 0.021 | 10.2 | 3.42 |
| | com-Orkut | 0.548 | 0.192 | 0.067 | – | 0.015 | 8.18 | 2.86 |
| | Twitter | 6.99 | 8.09 | 1.03 | – | 0.138 | 6.79 | 7.85 |
| BC | LiveJournal | 18.7 | 0.408 | 0.105 | 5.45 | 0.075 | 3.43 | 3.88 |
| | com-Orkut | 32.8 | 1.32 | 0.160 | 7.74 | 0.078 | 4.23 | 8.25 |
| | Twitter | 223 | 53.1 | 3.52 | 122 | 1.18 | 1.82 | 15.1 |

**Table 9.** Running times (in seconds) comparing the performance of algorithms implemented in Stinger (**ST**), LLAMA (**LL**), and Aspen. **A** is the parallel time using Aspen *without direction-optimization.* **(A(1))** is the one-thread time of Aspen, which is only relevant for comparing with Stinger's BC implementation. $\mathbf{A}^\dagger$ is the parallel time using Aspen *with direction-optimization.* **(ST/A)** is Aspen's speedup over Stinger and **(LL/A)** is Aspen's speedup over LLAMA.

# PaC-trees: An Update on Blocked Trees

PaC-trees build on Aspen by removing randomization.

| Graph | Vertices | Edges | Ours | Aspen | Aspen/Ours |
|---|---|---|---|---|---|
| *DBLP (DB)* | 425,957 | 2,099,732 | 0.0130 | 0.03409 | 2.62x |
| *YouTube (YT)* | 1,138,499 | 5,980,886 | 0.0412 | 0.0934 | 2.26x |
| *USA-Road (RU)* | 23,947,348 | 57,708,624 | 0.683 | 1.843 | 2.69x |
| *LiveJournal (LJ)* | 4,847,571 | 85,702,474 | 0.346 | 0.527 | 1.52x |
| *com-Orkut (CO)* | 3,072,627 | 234,370,166 | 0.727 | 0.893 | 1.22x |
| *Twitter (TW)* | 41,652,231 | 2,405,026,092 | 7.59 | 9.42 | 1.23x |
| *Friendster (FS)* | 65,608,366 | 3,612,134,270 | 14.6 | 19.1 | 1.30x |

**Table 4. Statistics about tested graphs and memory usage of PaC-tree and Aspen in GiB.**

| | | Aspen | | Ours | | | | |
|---|---|---|---|---|---|---|---|---|
| | Graph | FS | FS Time | No-FS | FS | FS/No-FS | FS Time | Aspen/Ours |
| **BFS** | *LiveJournal* | 21.7 | 3.82 | 19.8 | 17.5 | 1.13x | 1.38 | 1.24x |
| | *com-Orkut* | 15.3 | 2.35 | 14.5 | 12.4 | 1.16x | 1.12 | 1.23x |
| | *Twitter* | 138 | 37.8 | 125 | 112 | 1.11x | 12.5 | 1.23x |
| **MIS** | *LiveJournal* | 55.3 | 3.82 | 72.0 | 45.7 | 1.57x | 1.38 | 1.21x |
| | *com-Orkut* | 70.2 | 2.35 | 96.9 | 69.2 | 1.40x | 1.12 | 1.01x |
| | *Twitter* | 1022 | 37.8 | 1190 | 971 | 1.22x | 12.5 | 1.05x |
| **BC** | *LiveJournal* | 74.6 | 3.82 | 82.1 | 72.3 | 1.13x | 1.38 | 1.03x |
| | *com-Orkut* | 76.3 | 2.35 | 88.6 | 78.2 | 1.13x | 1.12 | 0.975x |
| | *Twitter* | 1150 | 37.8 | 2735 | 1030 | 2.65x | 12.5 | 1.11x |

**Table 5. Parallel running times (in milliseconds) for Aspen and our implementation.** We show the algorithm performance without flat snapshots (**No-FS**), with flat snapshots (**FS**), and the time to computing the flat snapshot (**FS Time**).

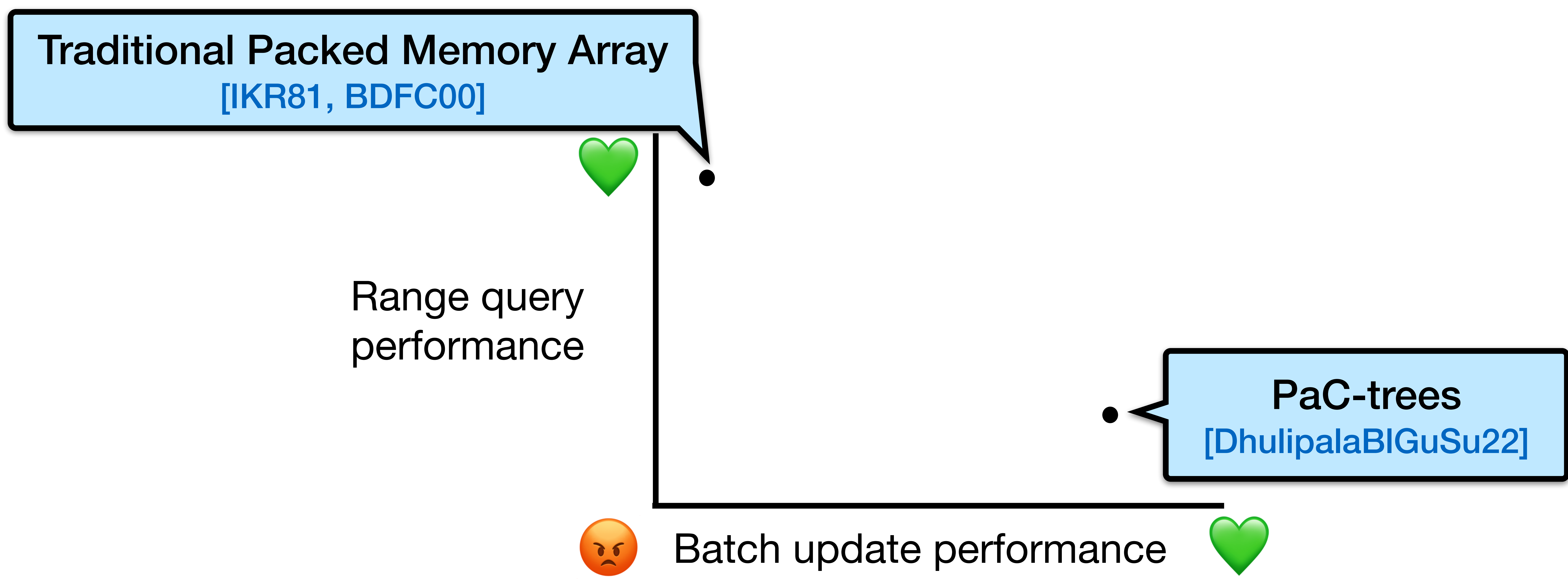
from "PaC-trees: Supporting Parallel and Compressed Purely-Functional Collections." Dhulipala, Blelloch, Gu, and Sun. PLDI 22.

# CPMA: An efficient batch-parallel set without pointers
# (Wheatman, Burns, Buluc, Xu - PPoPP 24)

# Update-Query Tradeoff in Batch-Parallel Sets

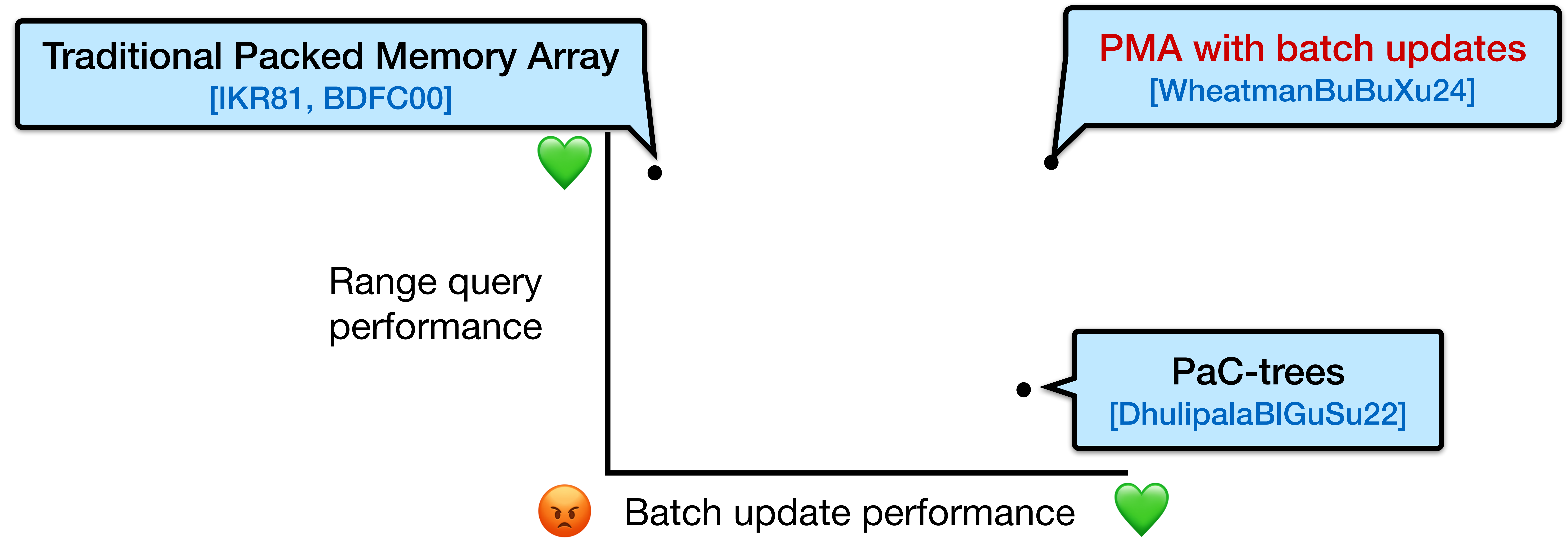
Existing batch-parallel data structures are built on **pointer-based structures** (e.g. trees, skip lists) [BarbuzziMiBiBo10, DhulipalaBlSh19, DhulipalaBlGuSu22, ErbKoSa14, FriasSi07, SunFeBl18, TsengDhBl19].

Pointer-based structures are **fast to update but slower to scan** compared to PMAs because they are not contiguous.



# Overcoming the Tradeoff with Cache-Optimized Data Structures

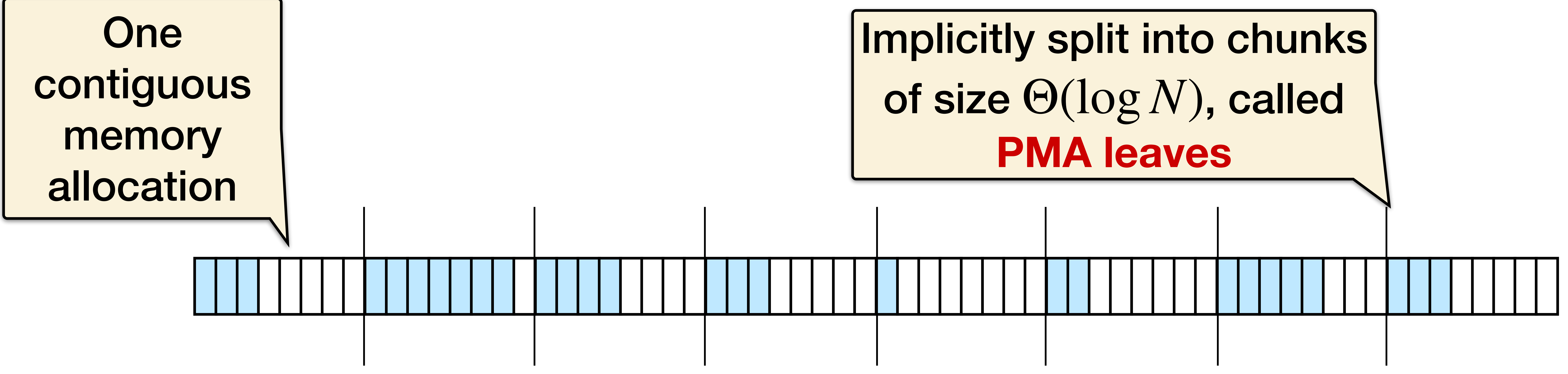
Although PMAs are asymptotically worse than trees for updates, their cache-friendliness enables them to achieve **faster updates in practice**.



# Recall: Packed Memory Array

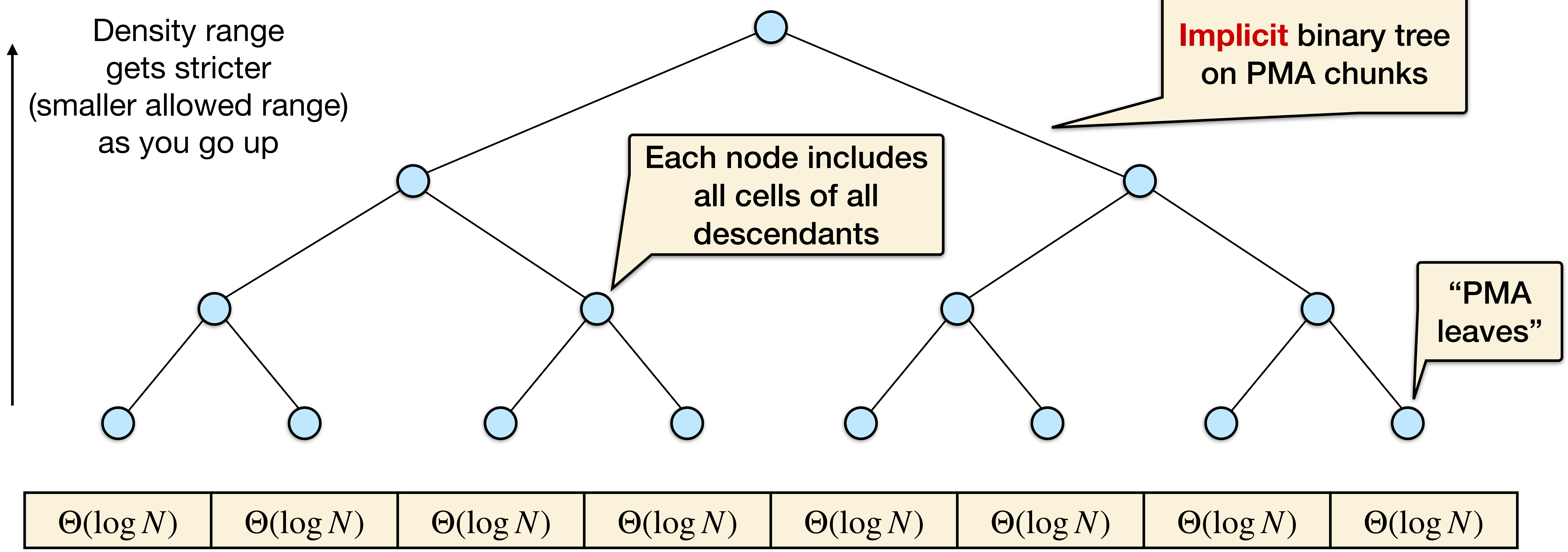
The Packed Memory Array (PMA) [ItaiKaRo81, BenderDeFa00] is a **cache-oblivious** ordered dictionary data structure that stores elements in a **contiguous array** with (a constant factor of) spaces for updatability.

That is, the PMA stores $N$ elements in m = $\Theta(N)$ cells.



# Recall: PMA Structure

The PMA maintains empty spaces according to density bounds, where the density is the **ratio of filled cells to total cells** per contiguous region.

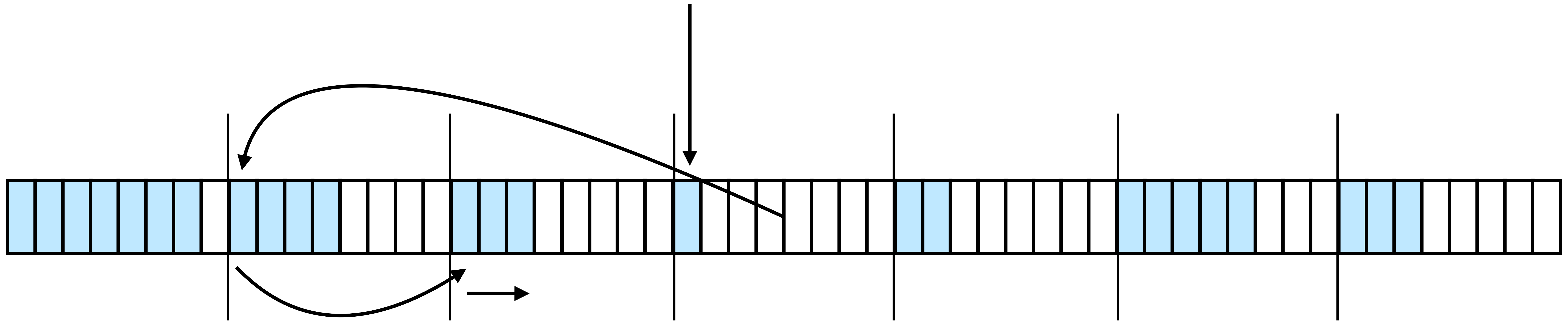


# Recall: Searching in a PMA

Searching a PMA involves a **binary search** on the first element of each PMA leaf.

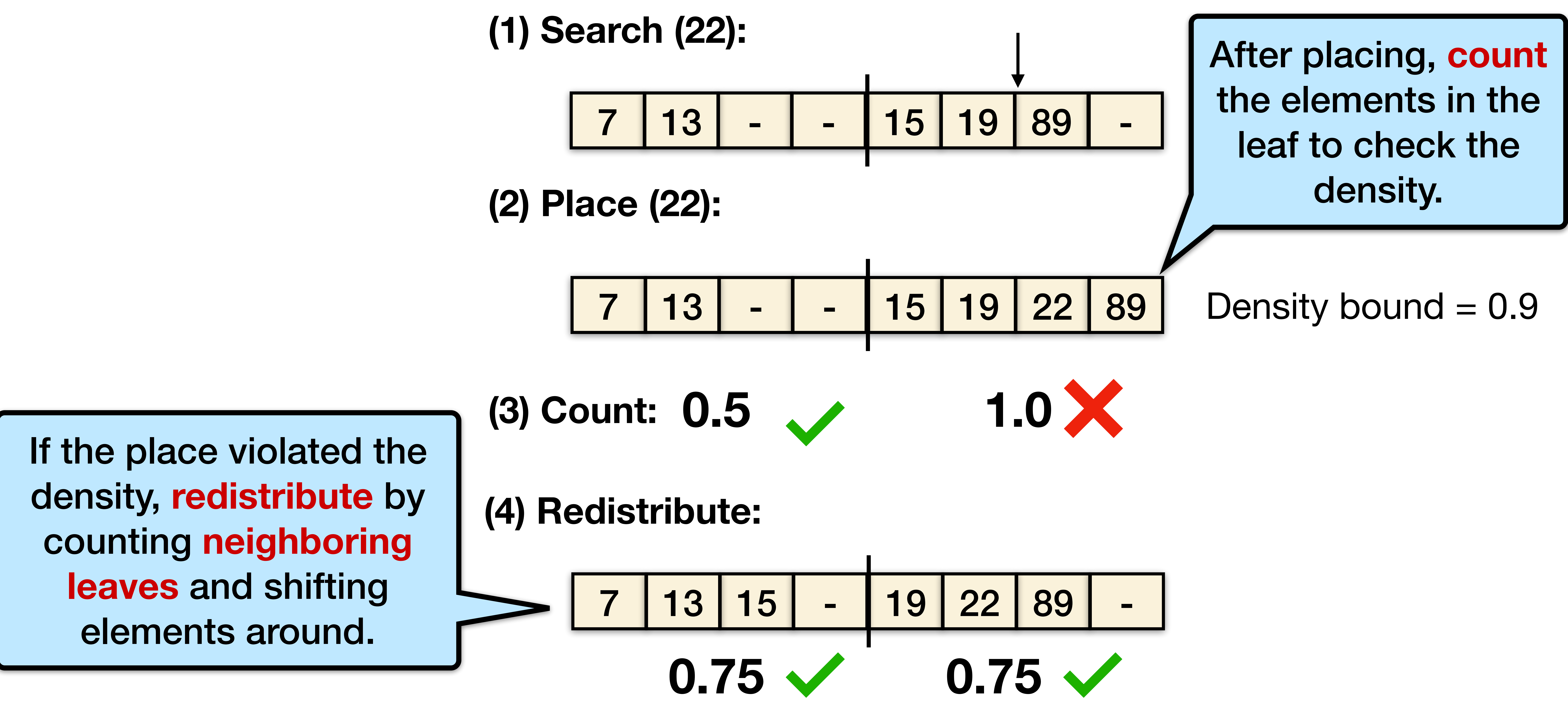
Once you reach the correct leaf, perform a **linear pass** through the chunk to look for the element.

The search costs $O(\log(N/\log(N)) + \log(N)/B) = O(\log(N)$ cache misses.



# PMA Insert Example

The PMA maintains density bounds during updates by **redistributing** elements after each update.



# PMA Parallel Batch Inserts - Overview

Batch merge

- Merge elements into the PMA

Counting nodes

- Determine which regions of the PMA need to be rebalanced

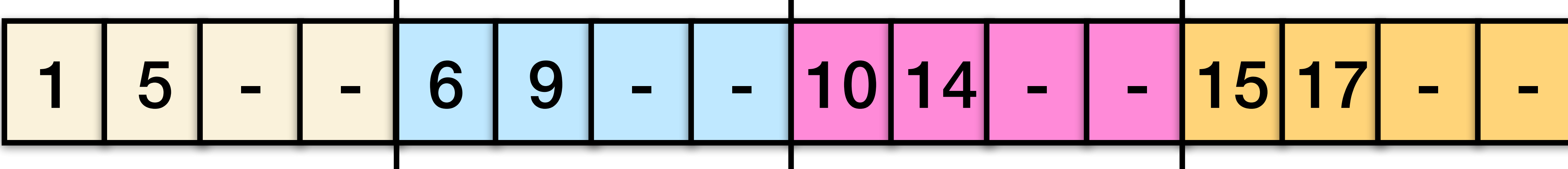
Redistribute nodes

- Rebalance the required regions

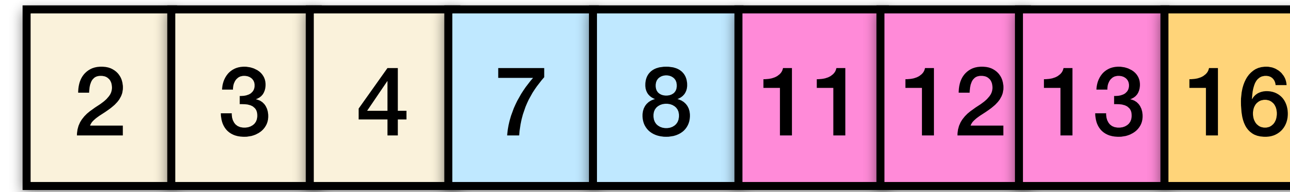
# PMA Batch Inserts - Batch Merge

Goal: Merge the elements in the batch into the correct positions in the PMA

**PMA**

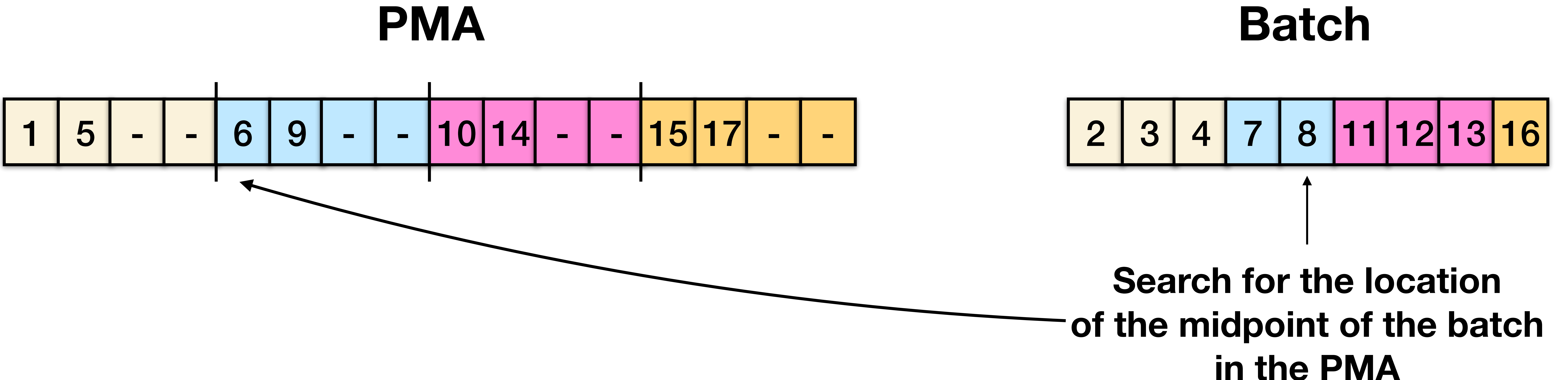



**Batch**



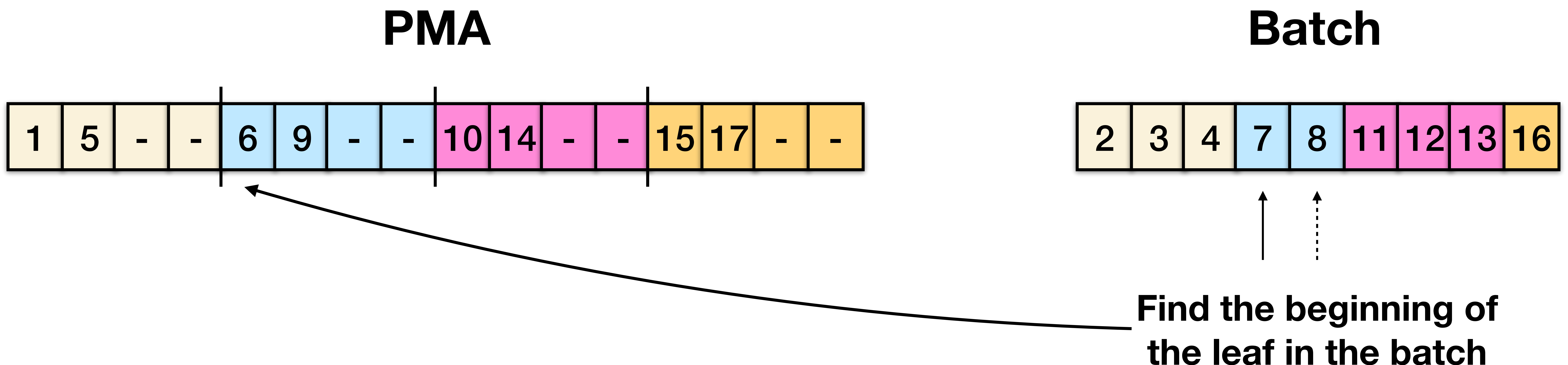
# PMA Batch Inserts - Batch Merge

Goal: Merge the elements in the batch into the correct positions in the PMA



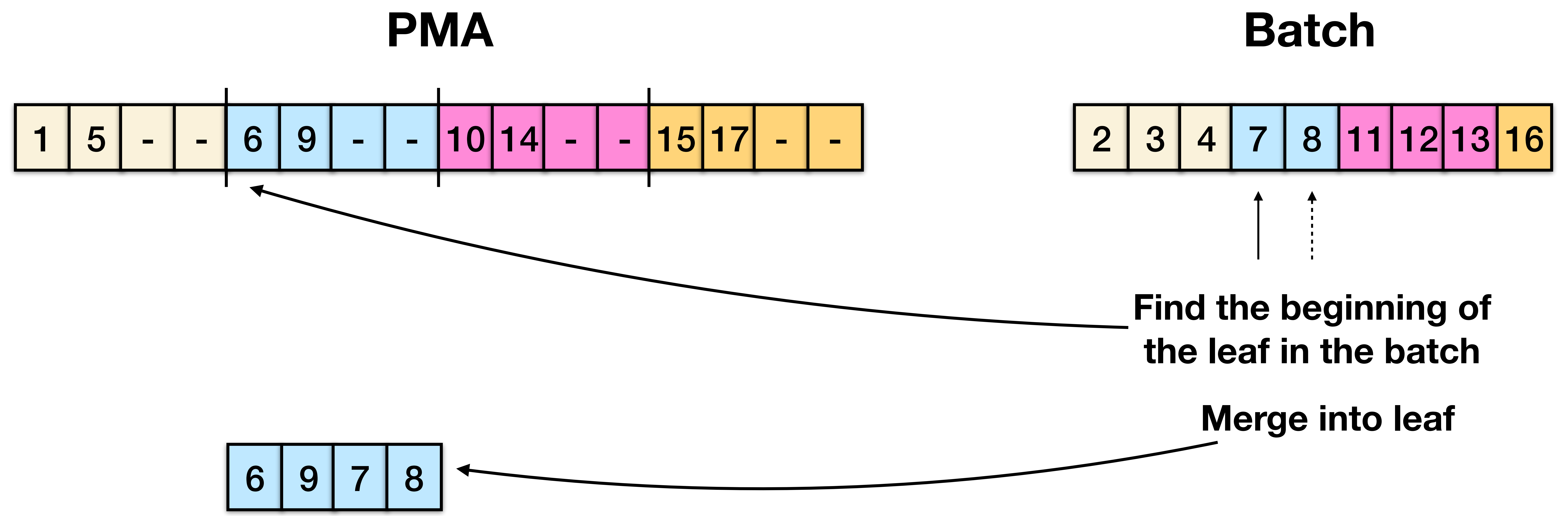
# PMA Batch Inserts - Batch Merge

Goal: Merge the elements in the batch into the correct positions in the PMA



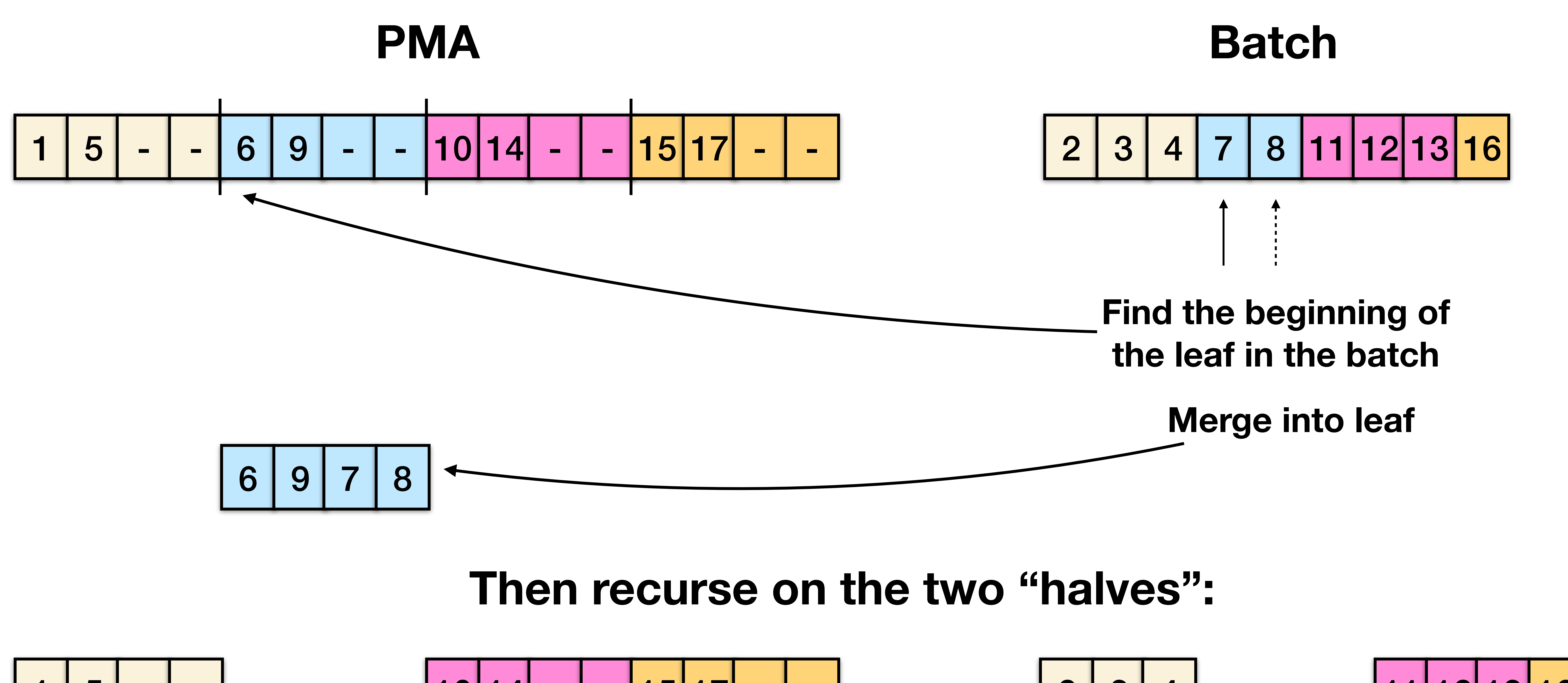
# PMA Batch Inserts - Batch Merge

Goal: Merge the elements in the batch into the correct positions in the PMA



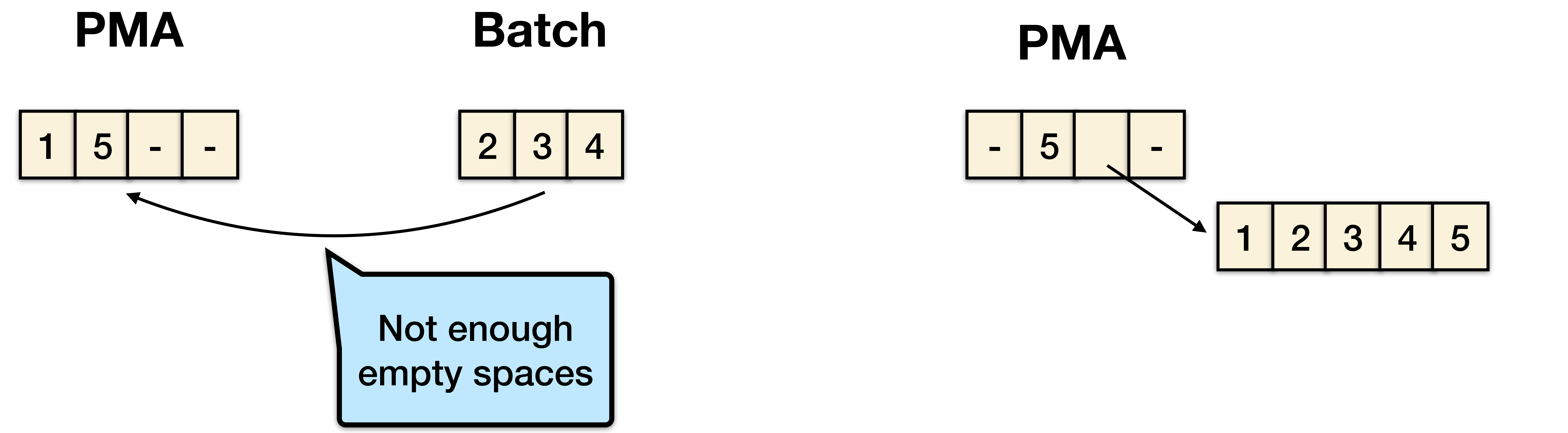
# PMA Batch Inserts - Batch Merge

Goal: Merge the elements in the batch into the correct positions in the PMA



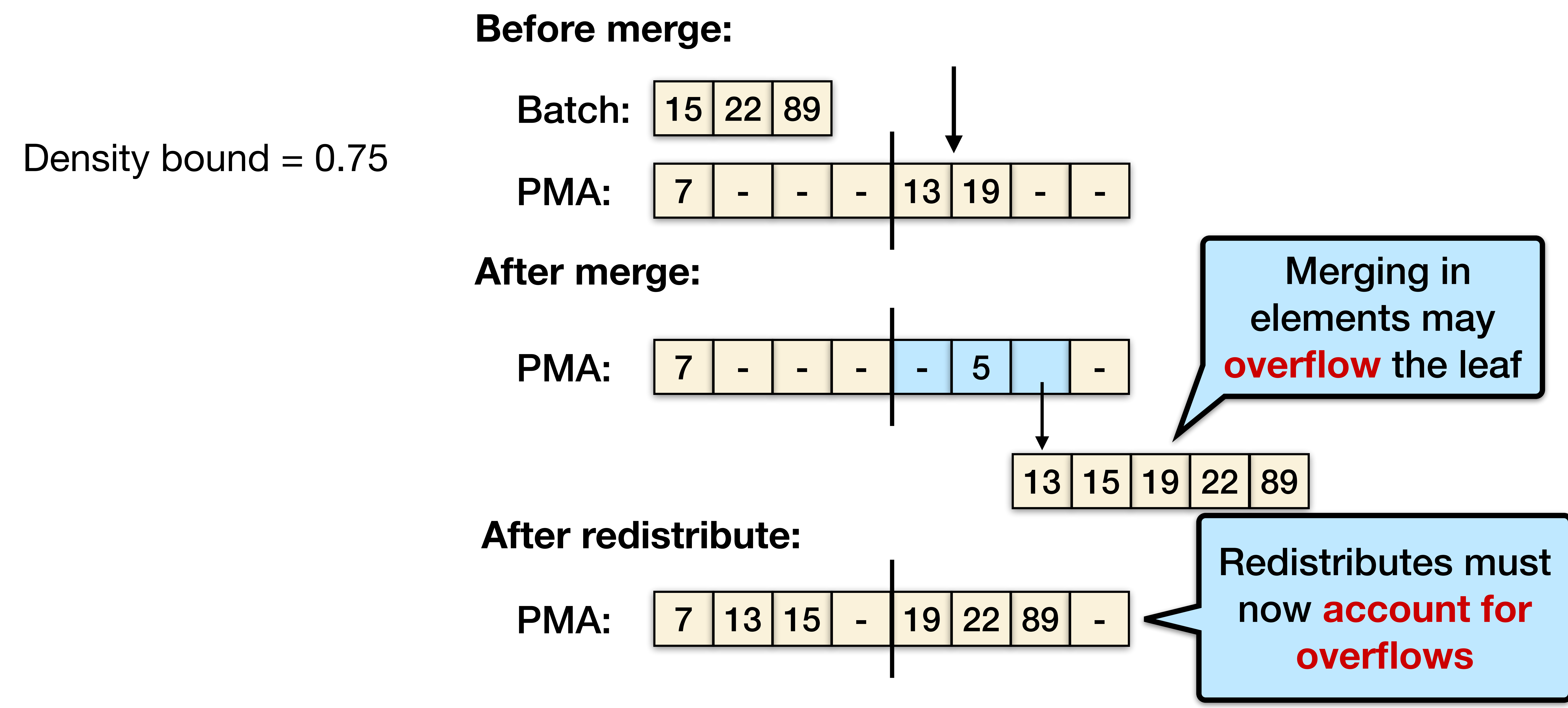
# PMA Batch Inserts - Handling Overflow

- Sometimes we need to **merge more elements than a leaf can fit**
- Since the other leaves are happening in parallel, we **cannot merge into neighboring leaves**
- Idea: temporarily store the elements **out of place** with a pointer and count



# Batch Updates Save Redundant Searches

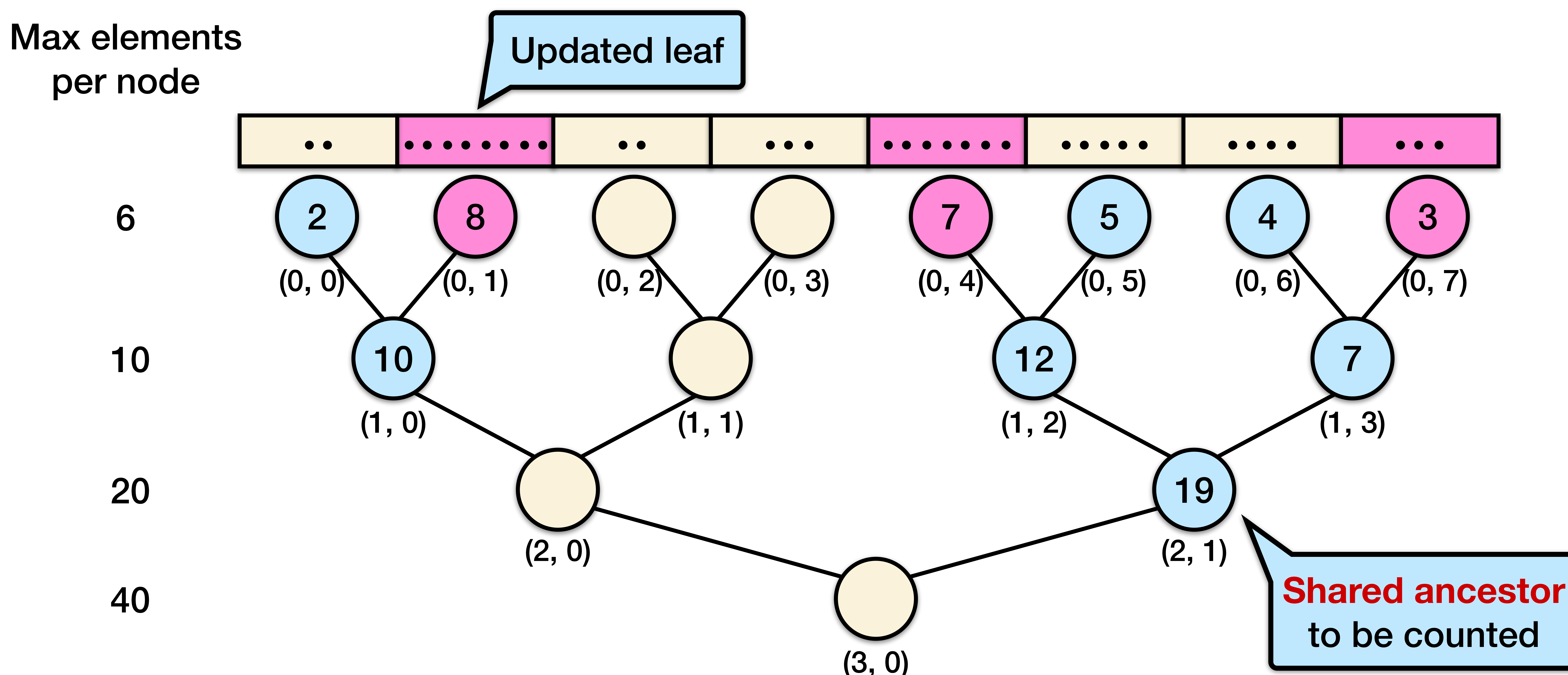
PMA batch updates save work by performing **only one search per PMA leaf** that elements are destined for (instead of performing one per element for point inserts).



# PMA Updates May Share Counting Work

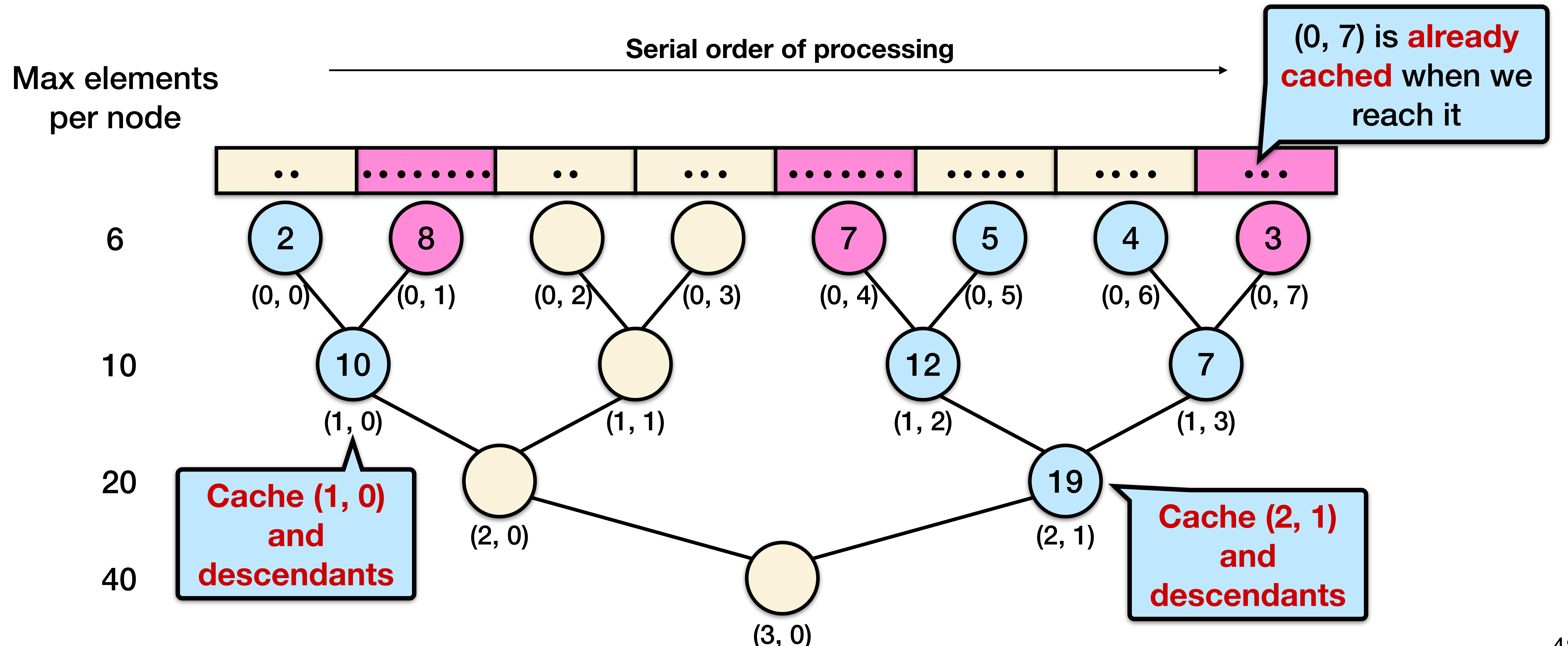
The PMA **implicit density tree determines which nodes need to be counted** before a redistribute.

**Multiple updated leaves** may share an ancestor in the density tree.



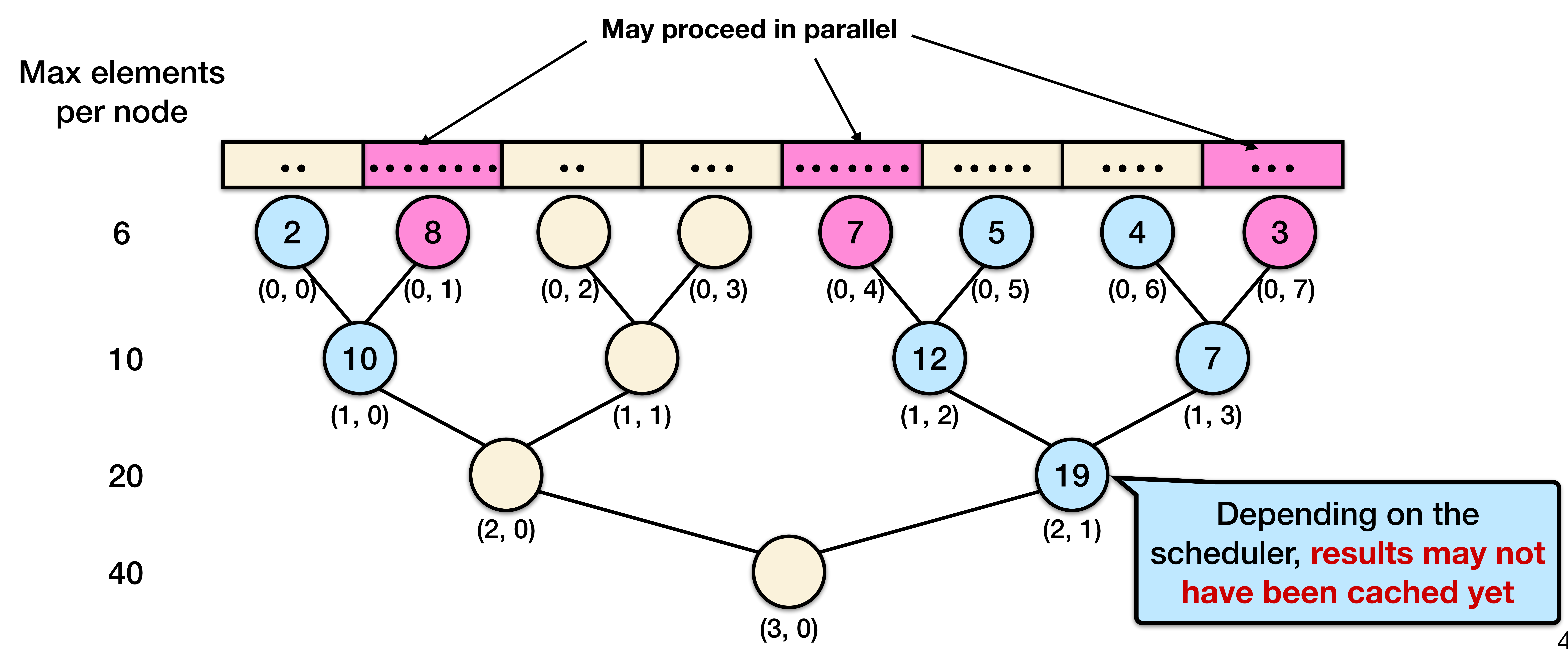
# Even Serial Batch Updates Can Save Redundant Counting

An **efficient serial algorithm** for performing multiple updates avoids redundant work by **caching results** from counting up elements in the PMA.



# Naively Counting in Parallel Batch Updates Takes Excess Work

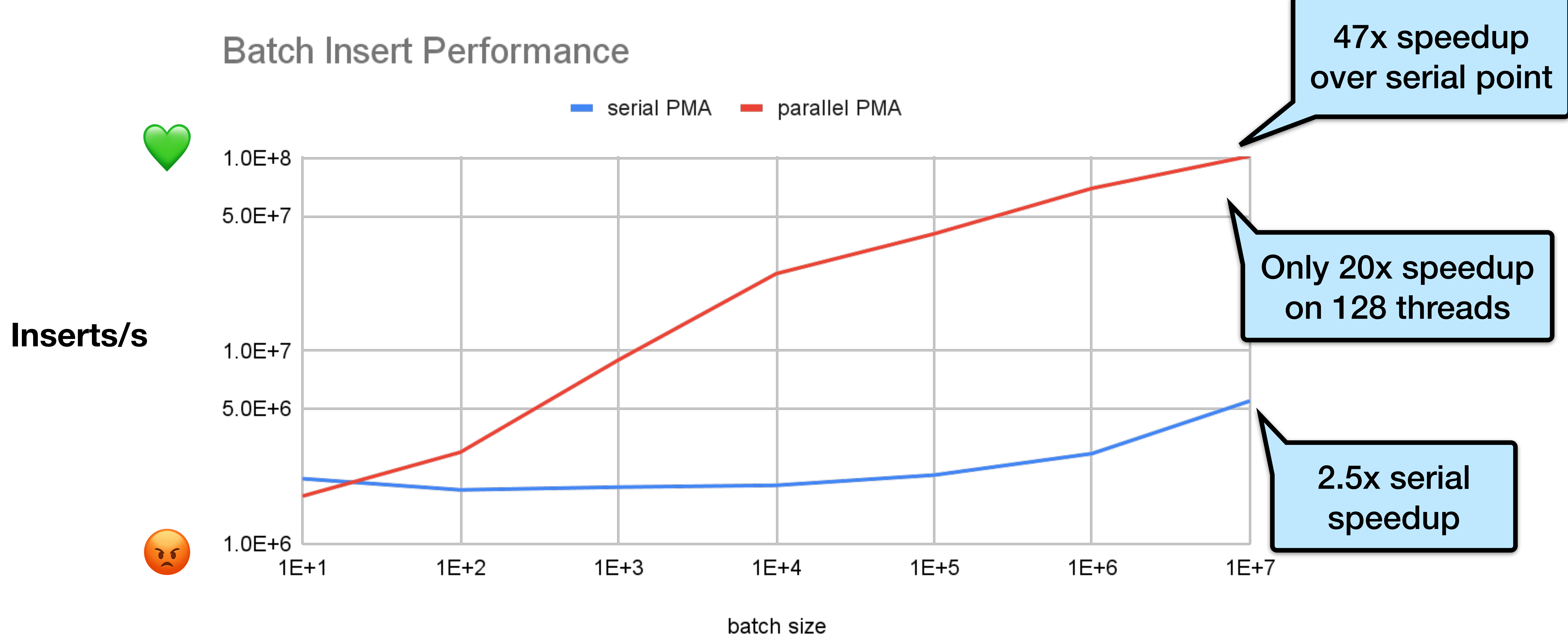
A **naive parallel algorithm** over the updates is not work-efficient because may **repeatedly count** the same cells.



# Work-Efficient PMA Batch Updates via Caching

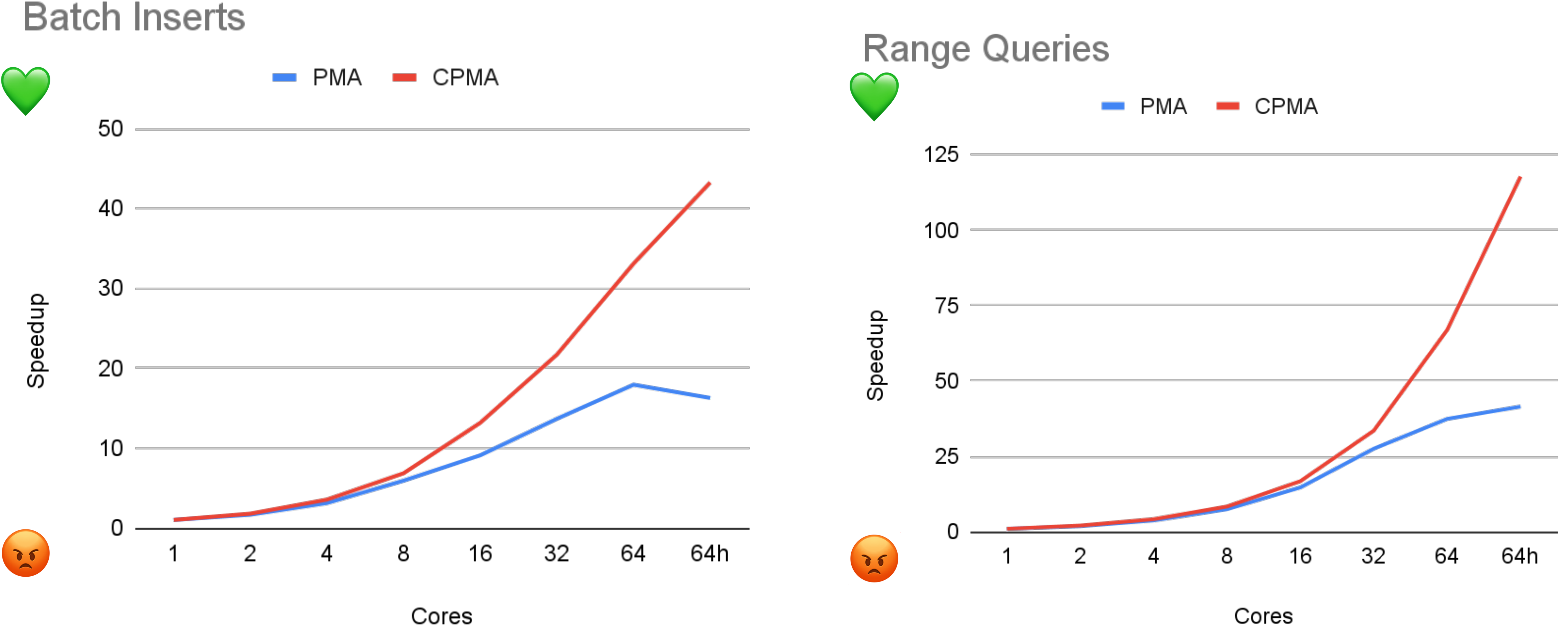
This paper introduces a **work-efficient counting algorithm** for PMA batch updates that counts every necessary cell exactly once.

# PMA Batch Insert Scaling

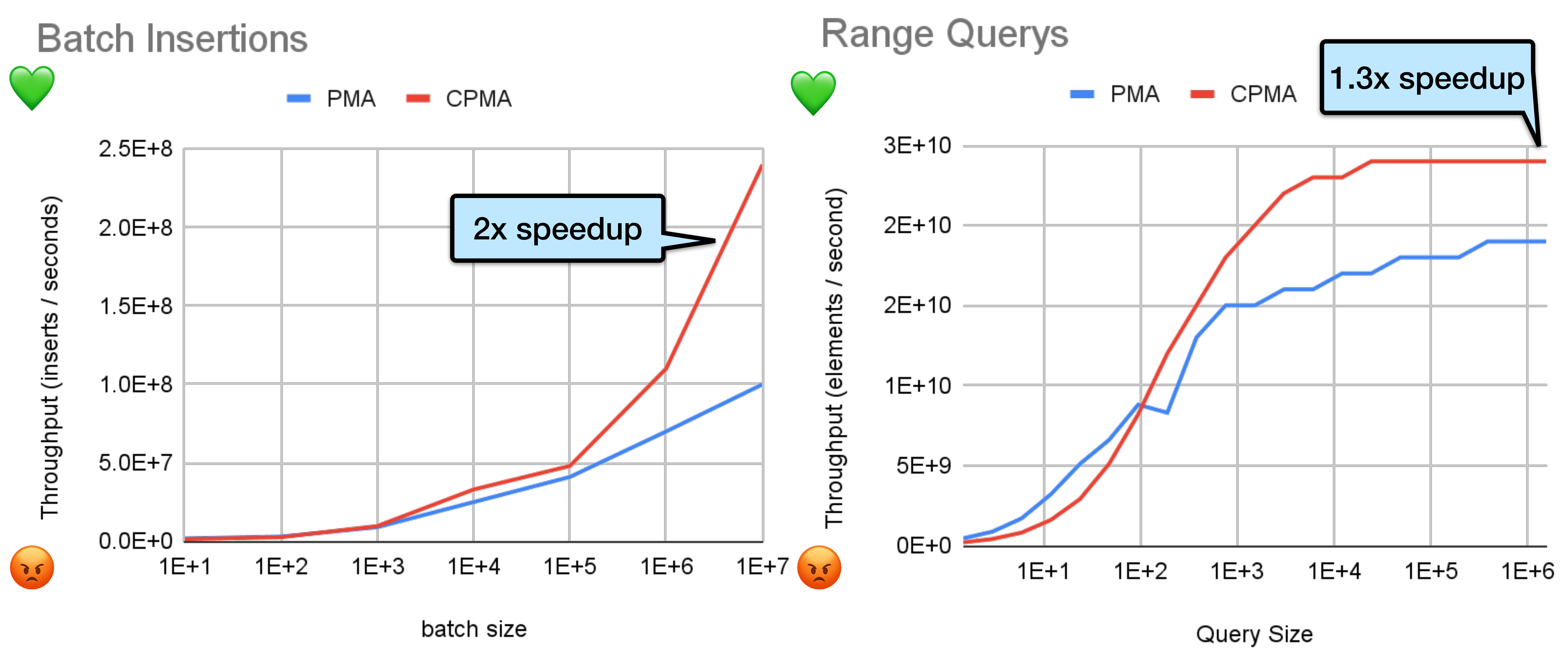


# Compression Improves Scalability

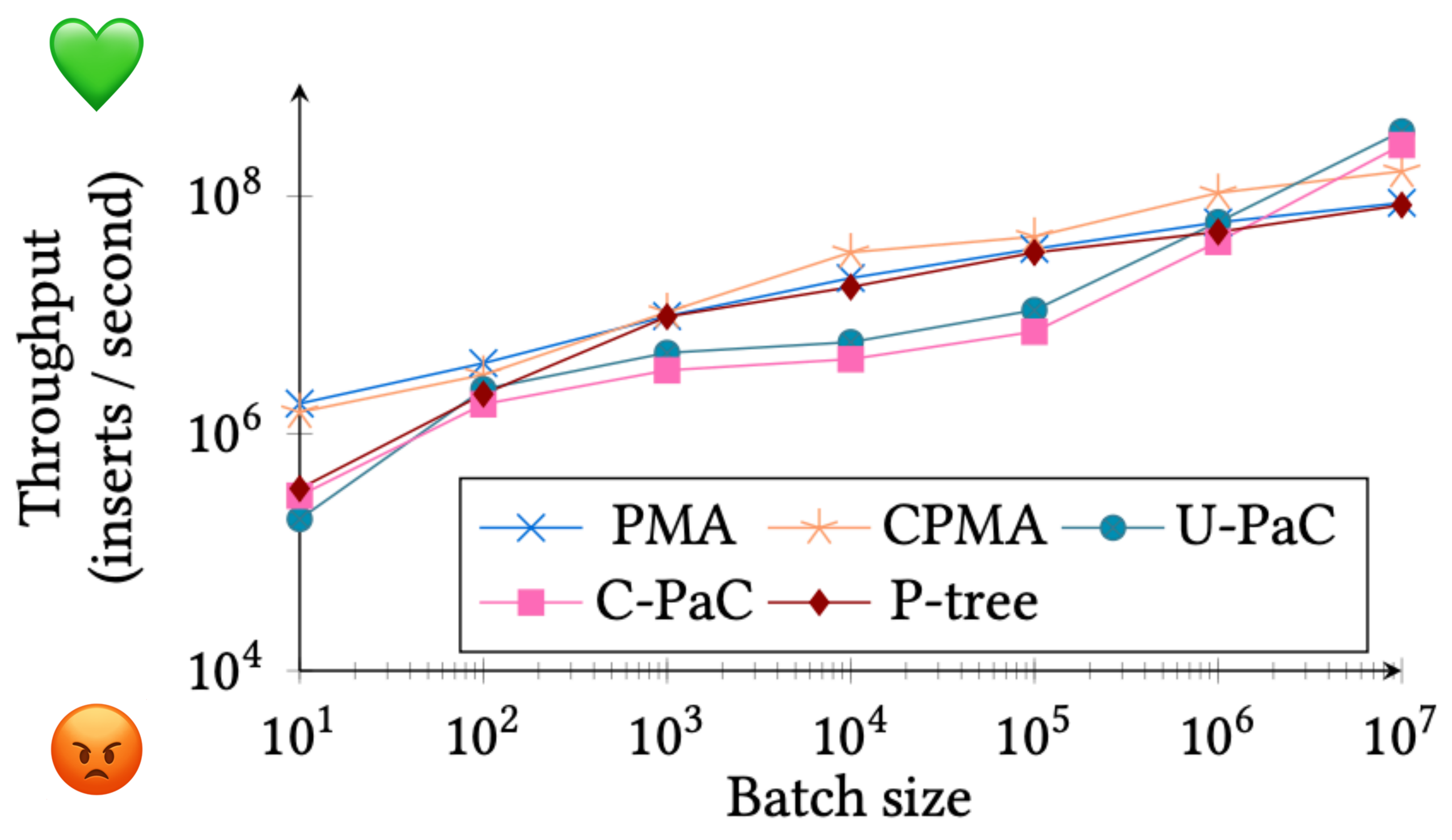
Compression can improve scalability by optimizing for memory bandwidth.

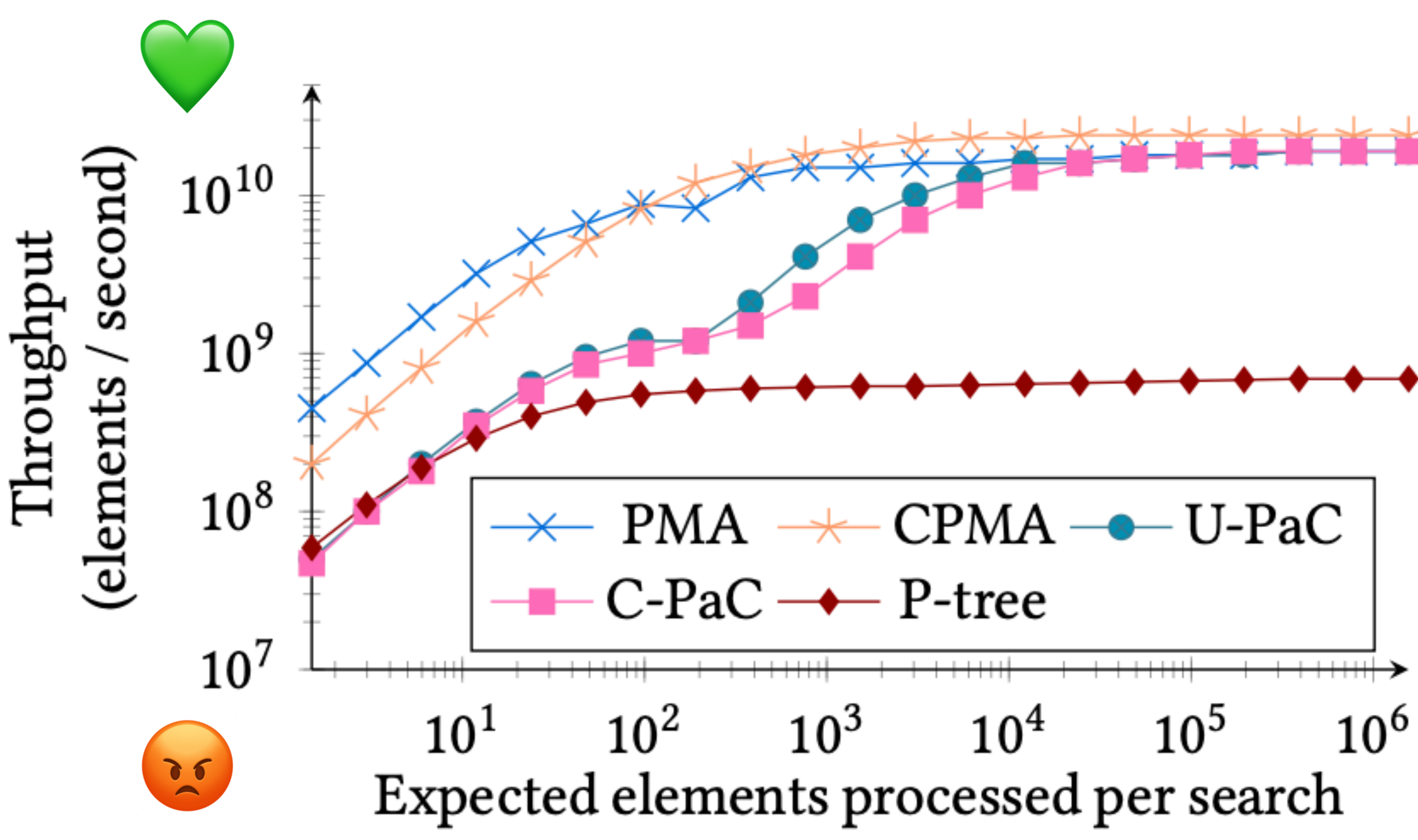


# Compression Improves Throutput



# Batch-Parallel PMA Results

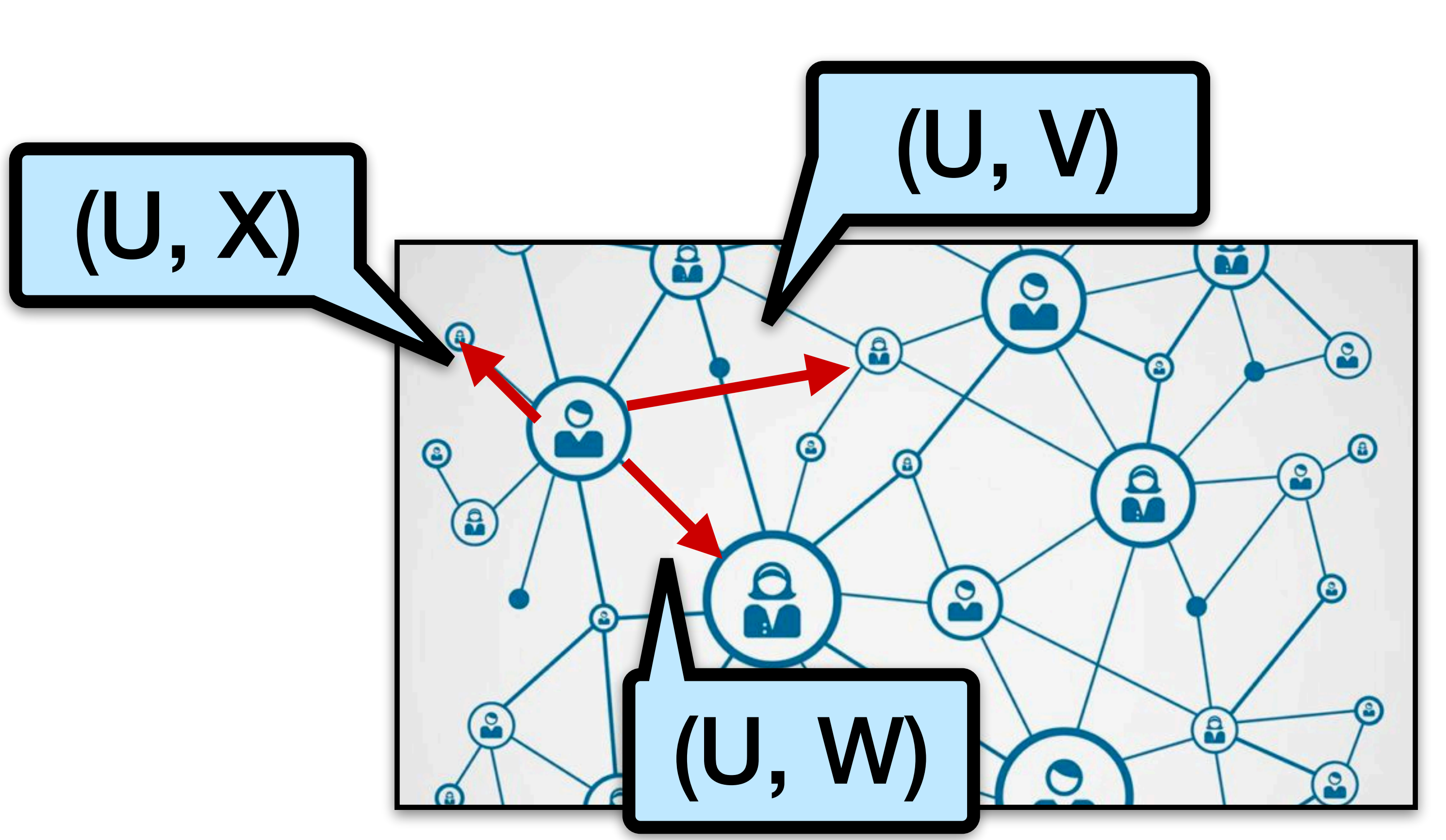


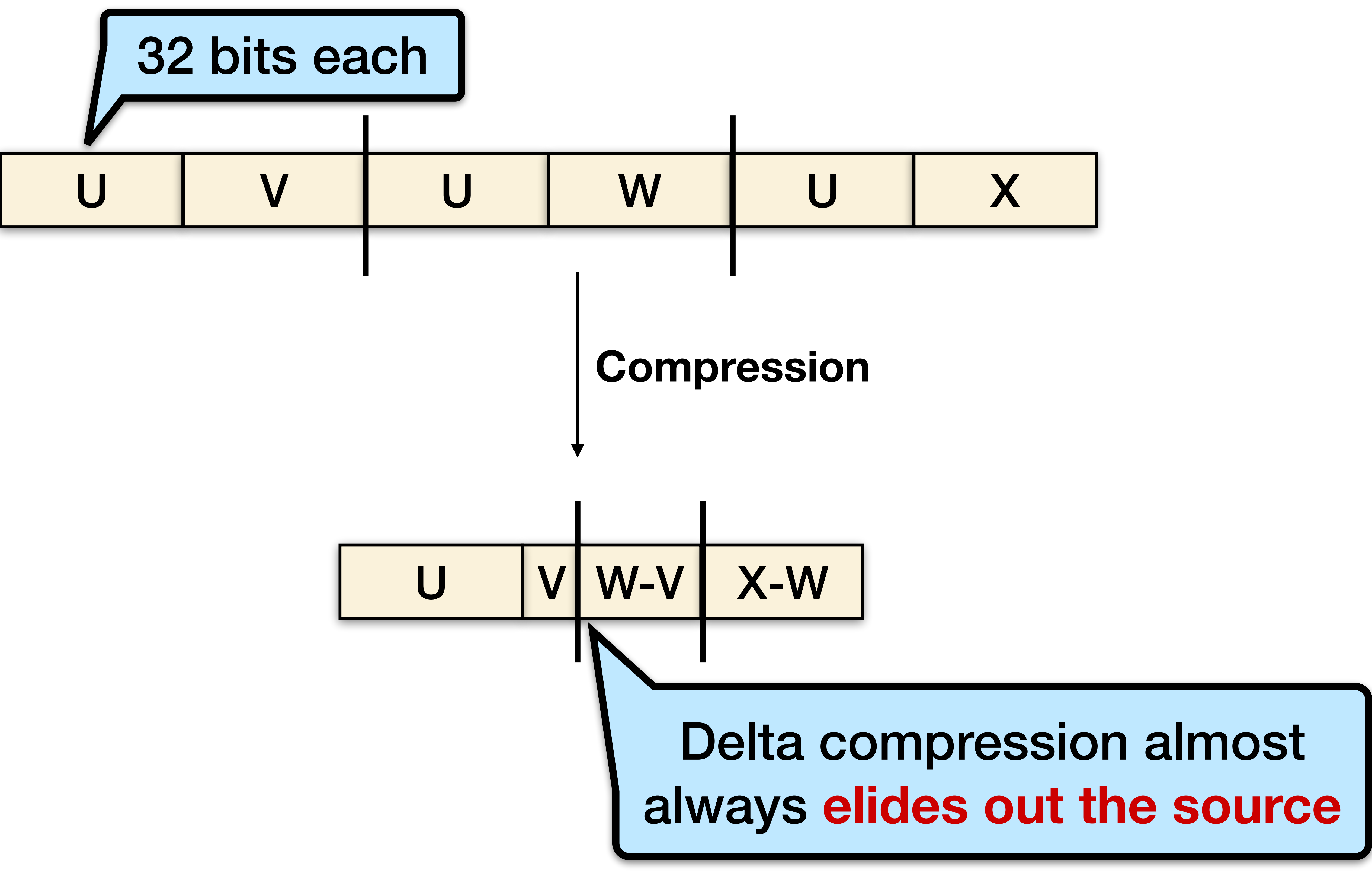



# Storing a Graph in One CPMA

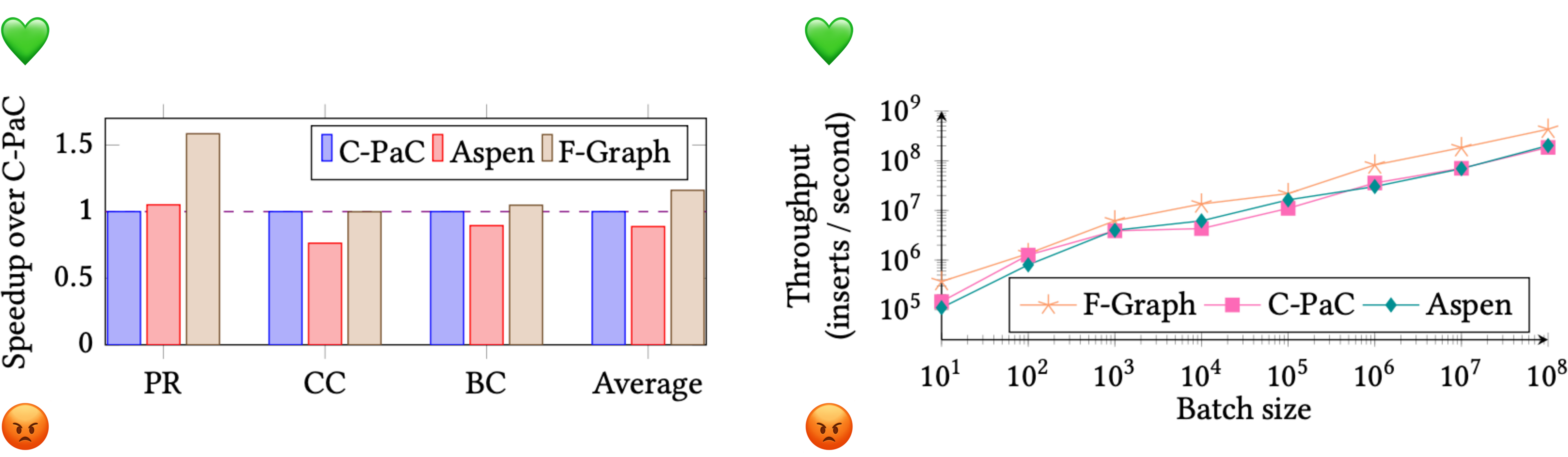
A graph can be represented as a sorted list of 64-bit edges where the upper 32 bits is the source and the lower 32 bits is the destination.

F-Graph stores this sorted list of edges in **a single CPMA**.



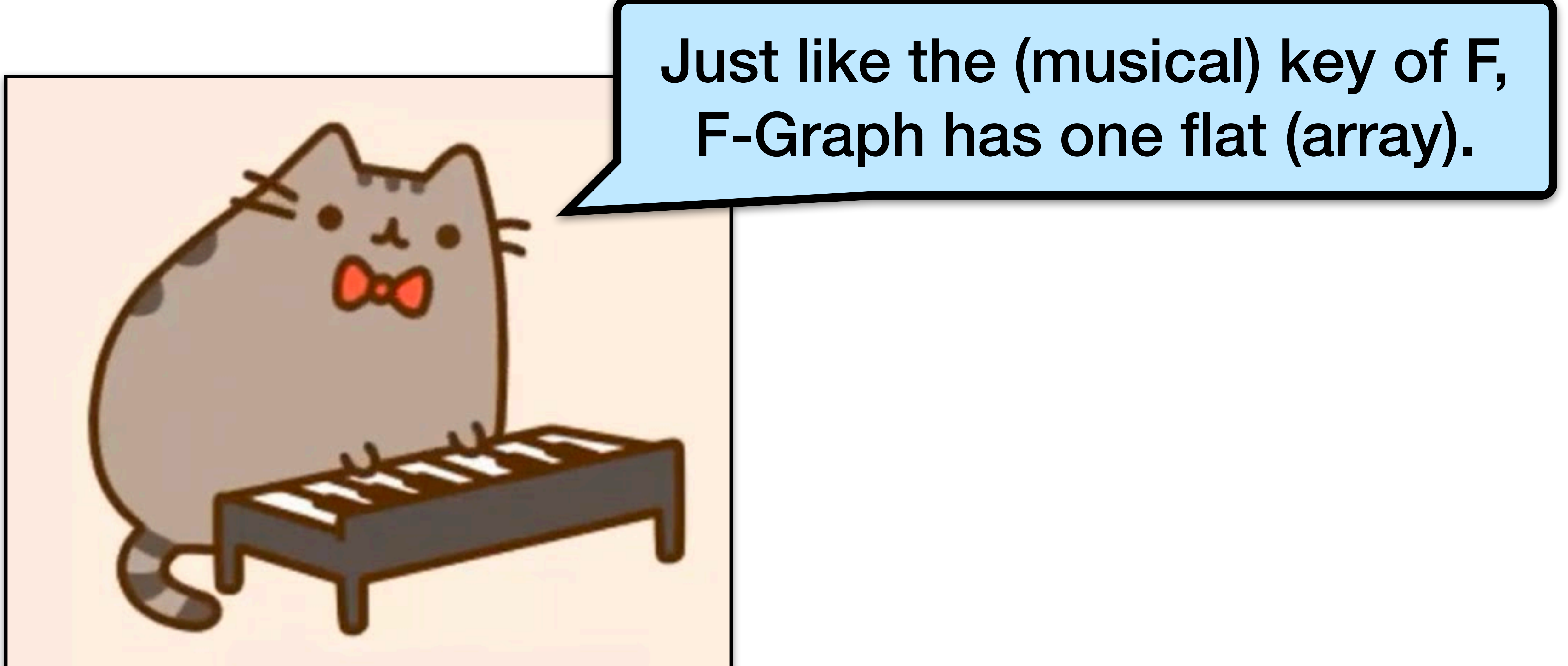
# Graph Results



# F-Graph: Overcoming Traditional Update/Scan Tradeoffs With Algorithmic Optimizations

Despite the theoretical prediction, CPMAs can **empirically overcome traditional update/scan tradeoffs** between PMAs and cache-optimized trees (PaC-trees) due to the PMA's locality.

F-Graph, a dynamic graph-processing system built on the CPMA, is **1.2x faster on graph algorithms** and on average **2x faster for graph updates compared to C-PaC,** a graph-processing system built on compressed trees.



# Summary

- Dynamic-graph data structures (containers) need to support **efficient algorithms and updates** to the graph.
- **Update and scan performance exhibit tension** due to locality concerns.
- Outside of just performance, graph data structures may have **other important features** (e.g., versioning, crash safety, etc.)
- In addition to the ones mentioned today, there have been **20+ years of papers on graph data structures** of all kinds (transactional, concurrent, distributed, GPU, etc).

# BACKUP

# Graph Sizes

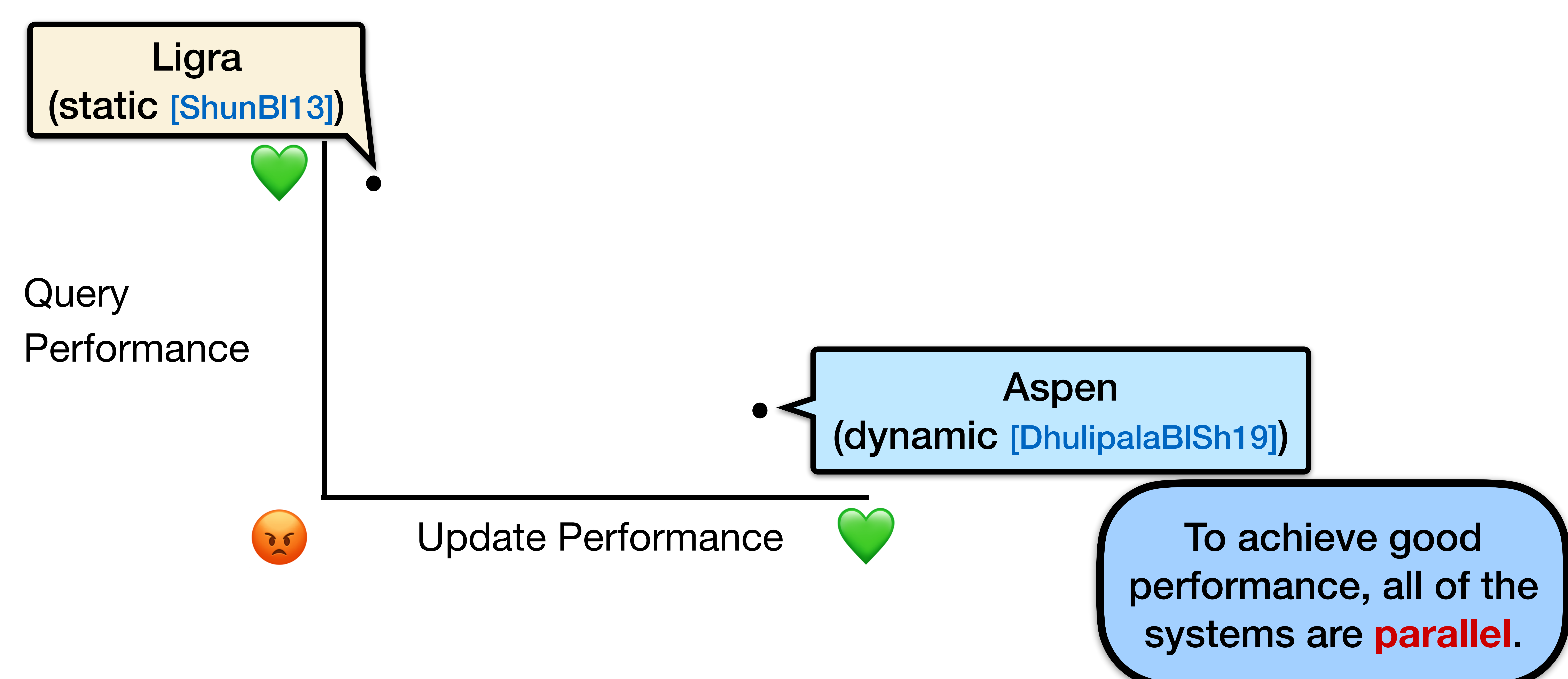
| Graph | Num. Vertices | Num. Edges | Avg. Deg. | Flat Snap. | Aspen Uncomp. | Aspen (No DE) | Aspen (DE) | Savings |
|---|---|---|---|---|---|---|---|---|
| *LiveJournal* | 4,847,571 | 85,702,474 | 17.8 | 0.0722 | 2.77 | 0.748 | 0.582 | 4.75x |
| *com-Orkut* | 3,072,627 | 234,370,166 | 76.2 | 0.0457 | 7.12 | 1.47 | 0.893 | 7.98x |
| *Twitter* | 41,652,231 | 2,405,026,092 | 57.7 | 0.620 | 73.5 | 15.6 | 9.42 | 7.80x |
| *ClueWeb* | 978,408,098 | 74,744,358,622 | 76.4 | 14.5 | 2271 | 468 | 200 | 11.3x |
| *Hyperlink2014* | 1,724,573,718 | 124,141,874,032 | 72.0 | 25.6 | 3776 | 782 | 363 | 10.4x |
| *Hyperlink2012* | 3,563,602,789 | 225,840,663,232 | 63.3 | 53.1 | 6889 | 1449 | 702 | 9.81x |

**Table 1.** Statistics about our input graphs, and memory usage using different formats in Aspen. **Flat Snap.** shows the amount of memory in GBs required to represent a flat snapshot of the graph. **Aspen Uncomp.**, **Aspen (No DE)**, and **Aspen (DE)** show the amount of memory in GBs required to represent the graph using uncompressed trees, Aspen without difference encoding of chunks, and Aspen with difference encoding of chunks, respectively. **Savings** shows the factor of memory saved by using Aspen (DE) over the uncompressed representation.

# Terrace: A hierarchical graph container for skewed dynamic graphs (Pandey, Wheatman, Xu, Buluc - SIGMOD 21)

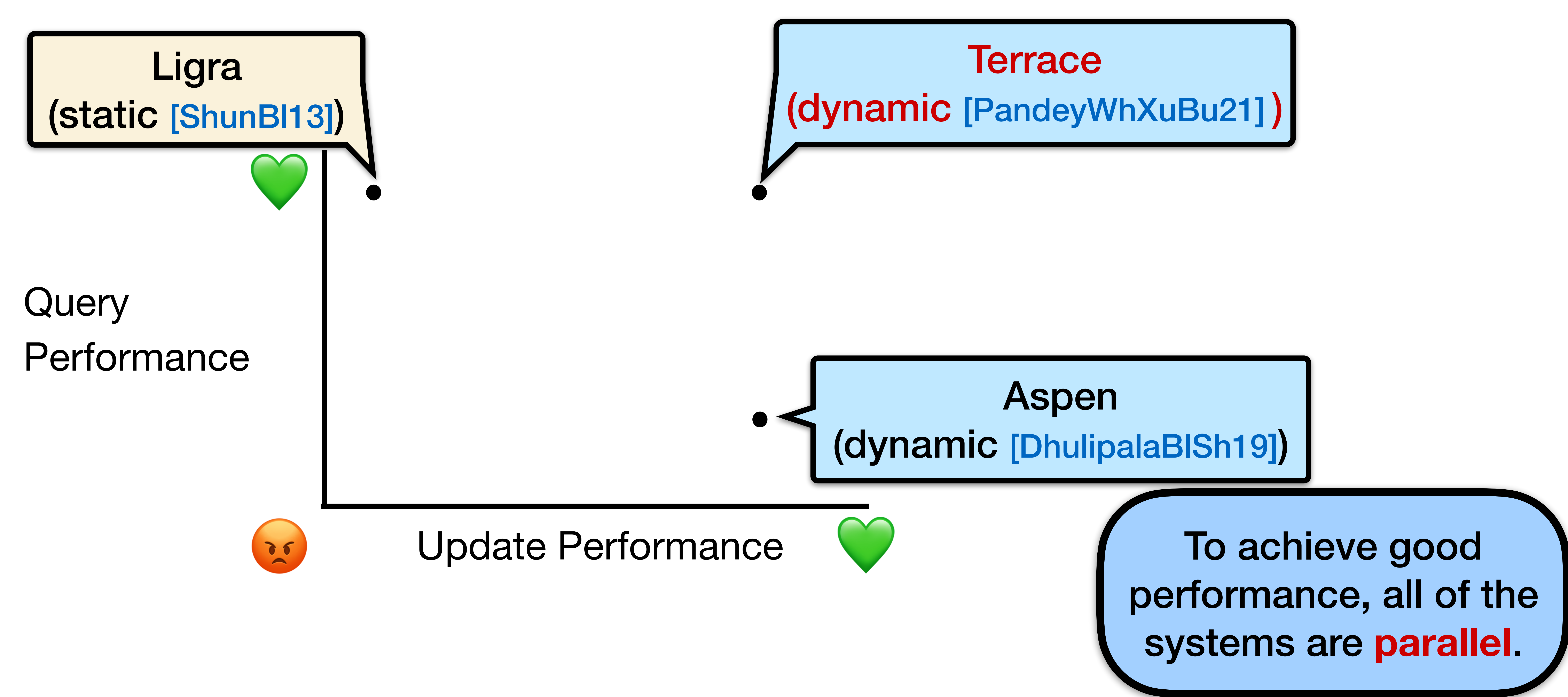
# Existing Graph Data Structures Trade Off Query and Update Performance

The commonly-held belief about graph data structures says that **query performance trades off with update performance** [EdigerMcRiBa12, KyrolaBlGu12, ShunBl13, MackoMaMaSe15, DhulipalaBlSh19, BusatoGrBoBa18, GreenBa16] due to **data representation** choices.



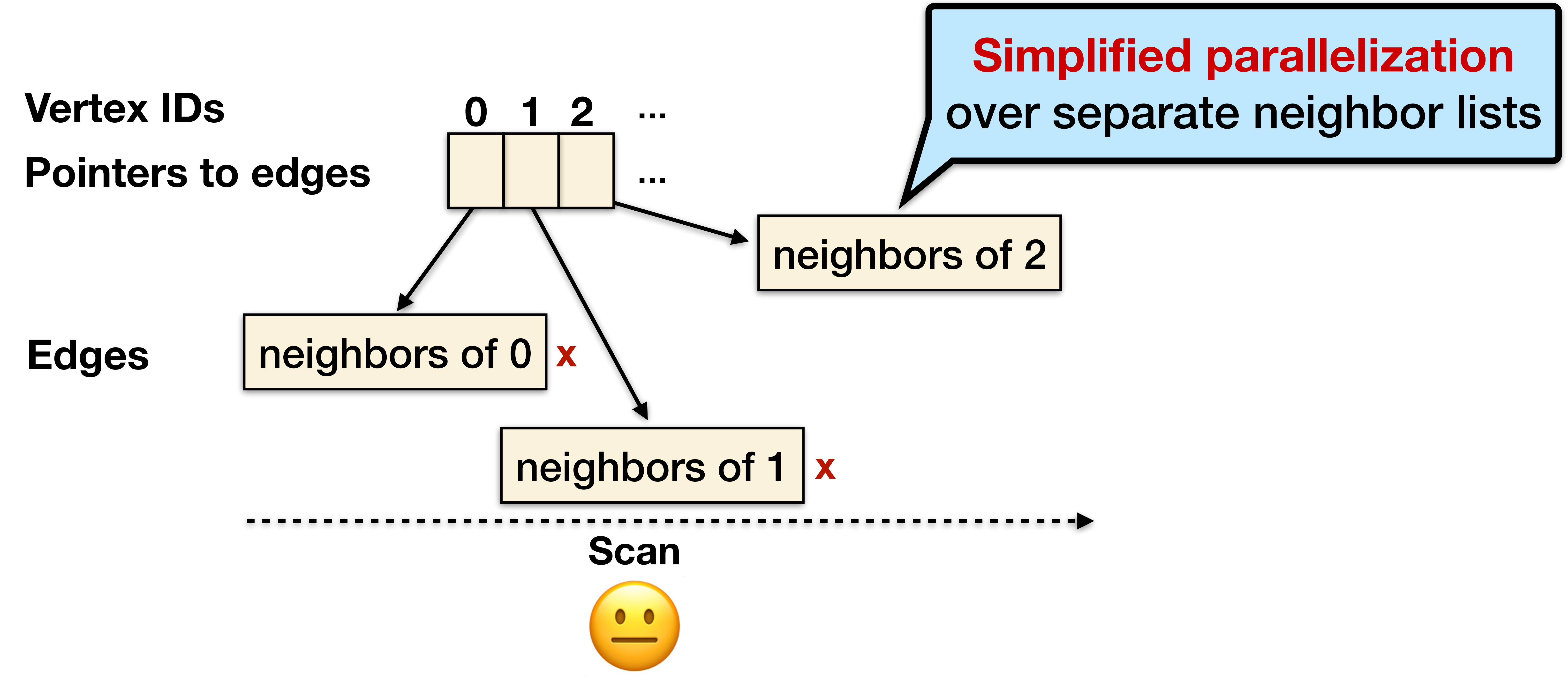
# Terrace: Overcoming the Query-Update Tradeoff with Locality-Optimized Data Structure Design

Terrace **achieves good query and update performance** by using data structures that enhance spatial locality.



# Understanding Opportunities for Locality in Separate Per-Vertex Data Structure Design

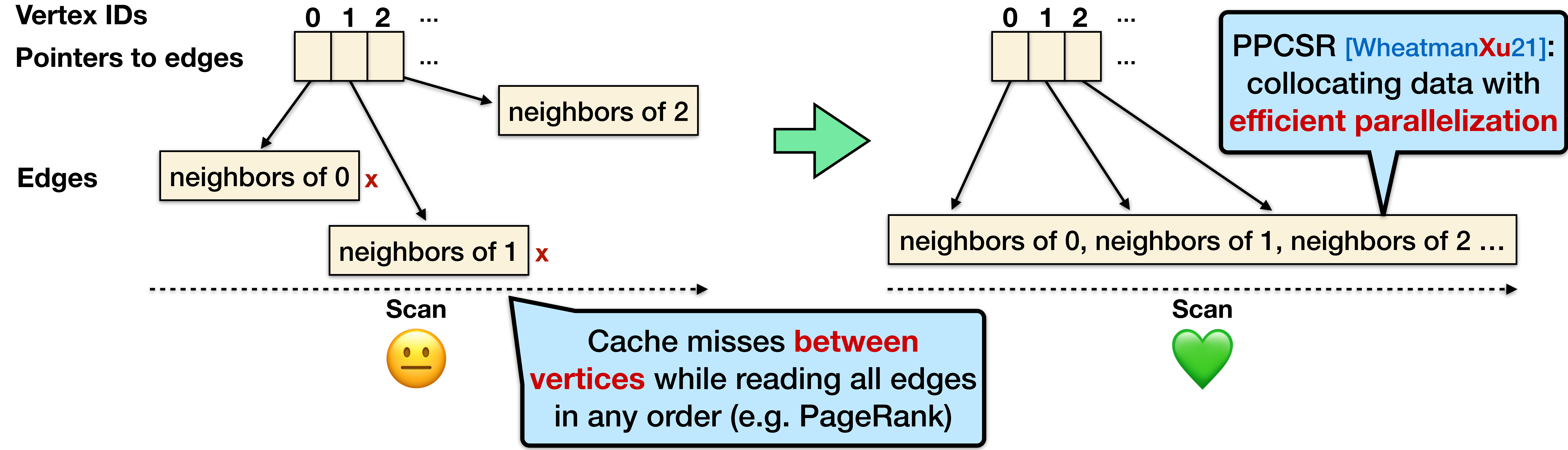
Existing dynamic graph systems optimize for parallelism first with **separate per-vertex data structures** e.g., trees [DhulipalaBlSh19], adjacency lists [EdigerMcRiBa12], and others [KyrolaBlGu12, BusatoGrBoBa18, GreenBa16].




Weakness: Separating the data structures **disrupts locality**.

# Enhancing Spatial Locality by Collocating Neighbor Data Structures

Idea: Collocate previously separate per-vertex data structures in the same data structure, which **avoids cache misses** when traversing edges in order.




**Question:** Do these misses actually affect performance, or are they a low-order term?

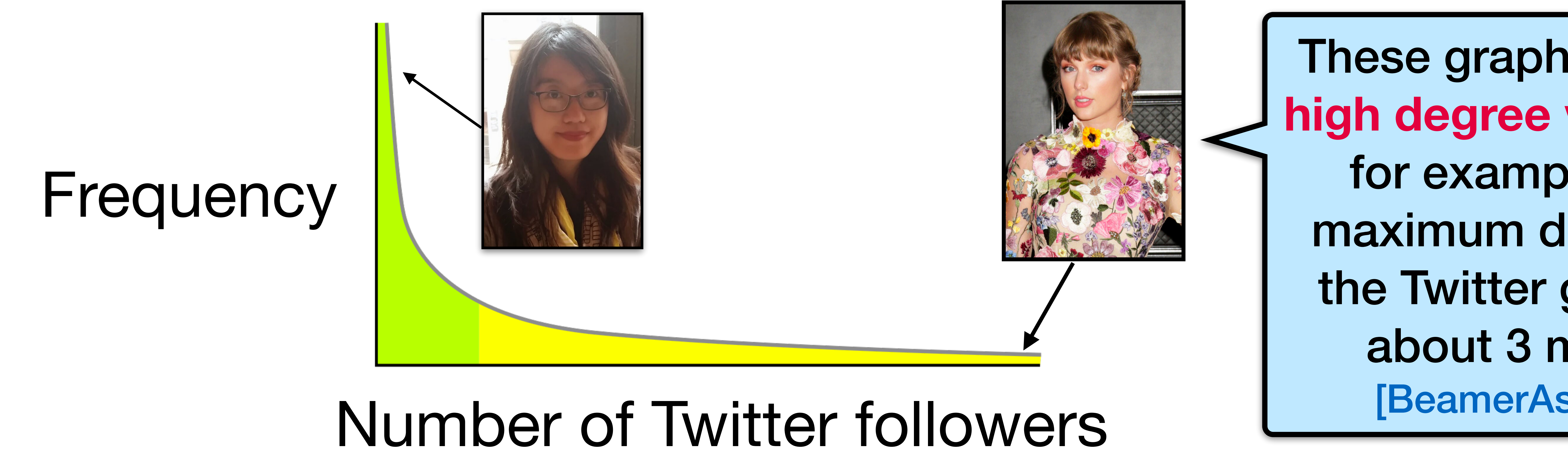
[WheatmanXu21] Wheatman and **Xu**. "A Parallel Packed Memory Array to Store Dynamic Graphs." ALENEX '21.

# Collocating Neighbor Data Structures Exploits Naturally-Occurring Skewness in Graphs

Collocating neighbor lists improves performance because real-world dynamic graphs, e.g., social network graphs, often follow a **skewed** (e.g., power-law) distribution with a **few high-degree vertices and many low-degree vertices** [BarabasiAl99].

Example power law:

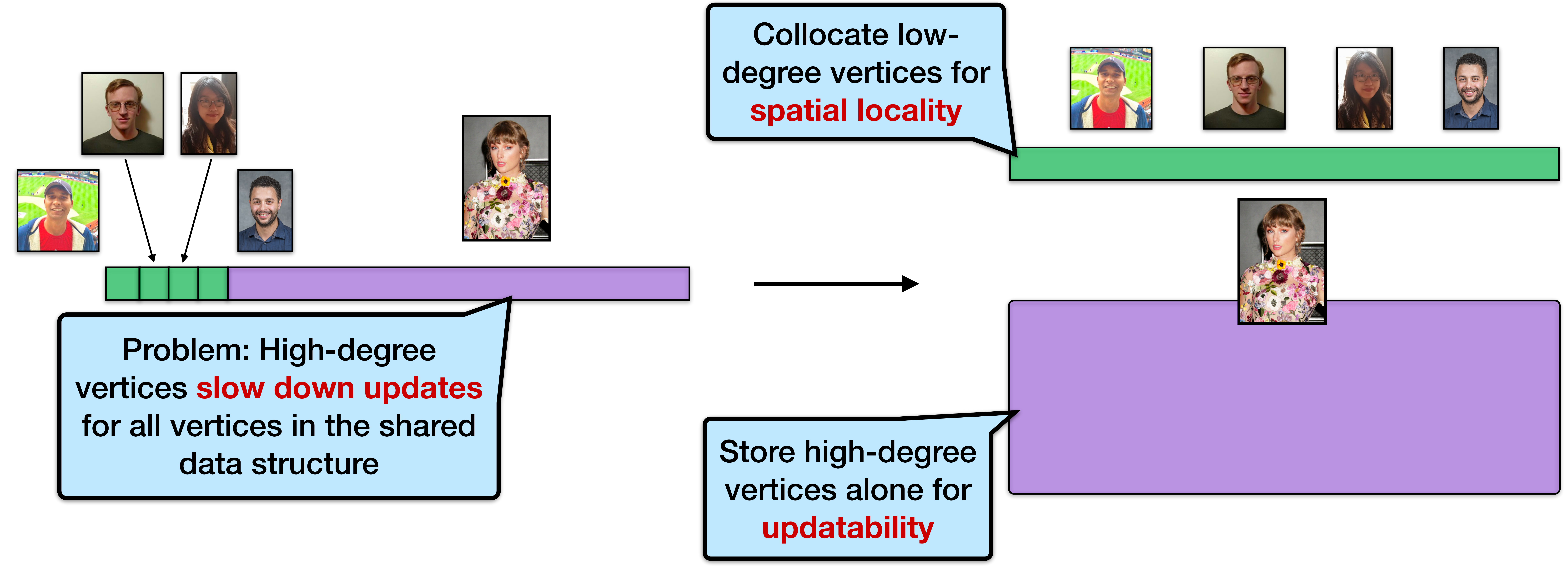
| Graph | % < 10 neighbors | % < 1000 neighbors |
|---|---|---|
| **Twitter** [BeamerAsPa15] | 64.6 | 99.5 |




These graphs exhibit **high degree variance**: for example, the maximum degree in the Twitter graph is about 3 million [BeamerAsPa15]

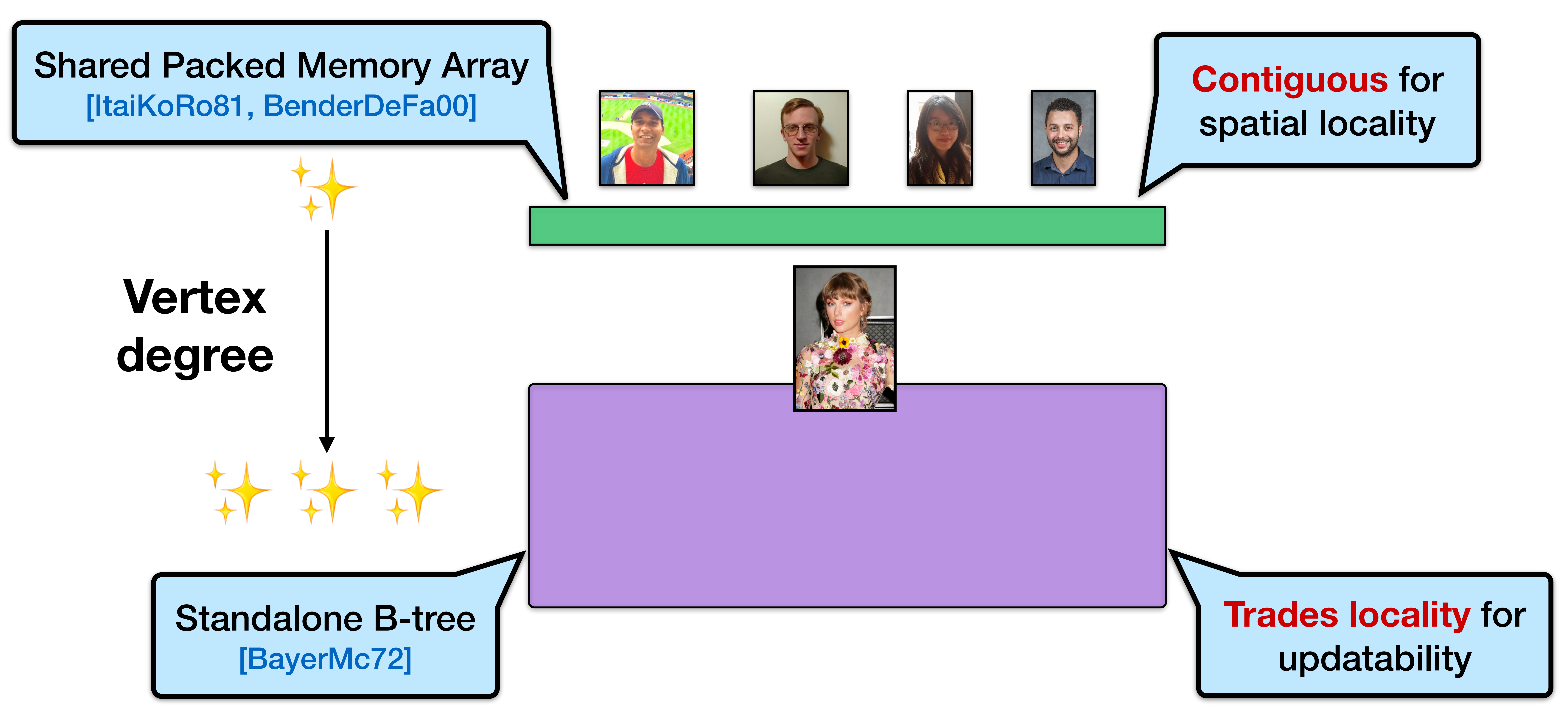
# Insight: Further Optimizing for Locality with a Hierarchical Skew-Aware Design

Next step: **refine the solution** with a **hierarchical design** that takes advantage of skewness while maintaining locality as much as possible.



# Implementing the Hierarchical Skew-Aware Design with Cache-Optimized Data Structures

Terrace implements the skew-aware hierarchical design with **cache-friendly data structures** that store vertex neighbors **depending on vertex degree**.

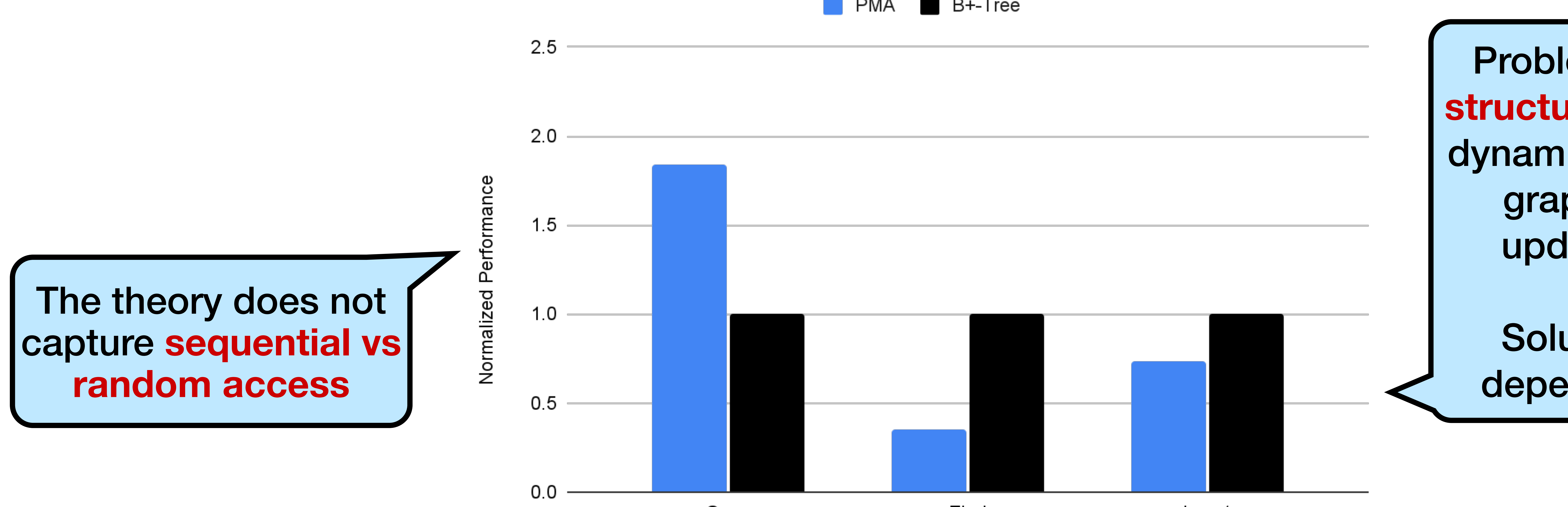


# Selecting Data Structures for Dynamic Graphs

In theory, B-trees [BayerMc72] **asymptotically dominate** Packed Memory Arrays (PMA) [ItaiKoRo81, BenderDeFa00] in the classical external-memory model [AggarwalVi88].

Given a cache block size $B$ and input size $N$, B-trees and PMAs take $\Theta(N/B)$ **block transfers** to scan.

B-tree inserts take $O(\log_B(N))$ transfers, while PMA inserts take $O(\log^2(N))$.

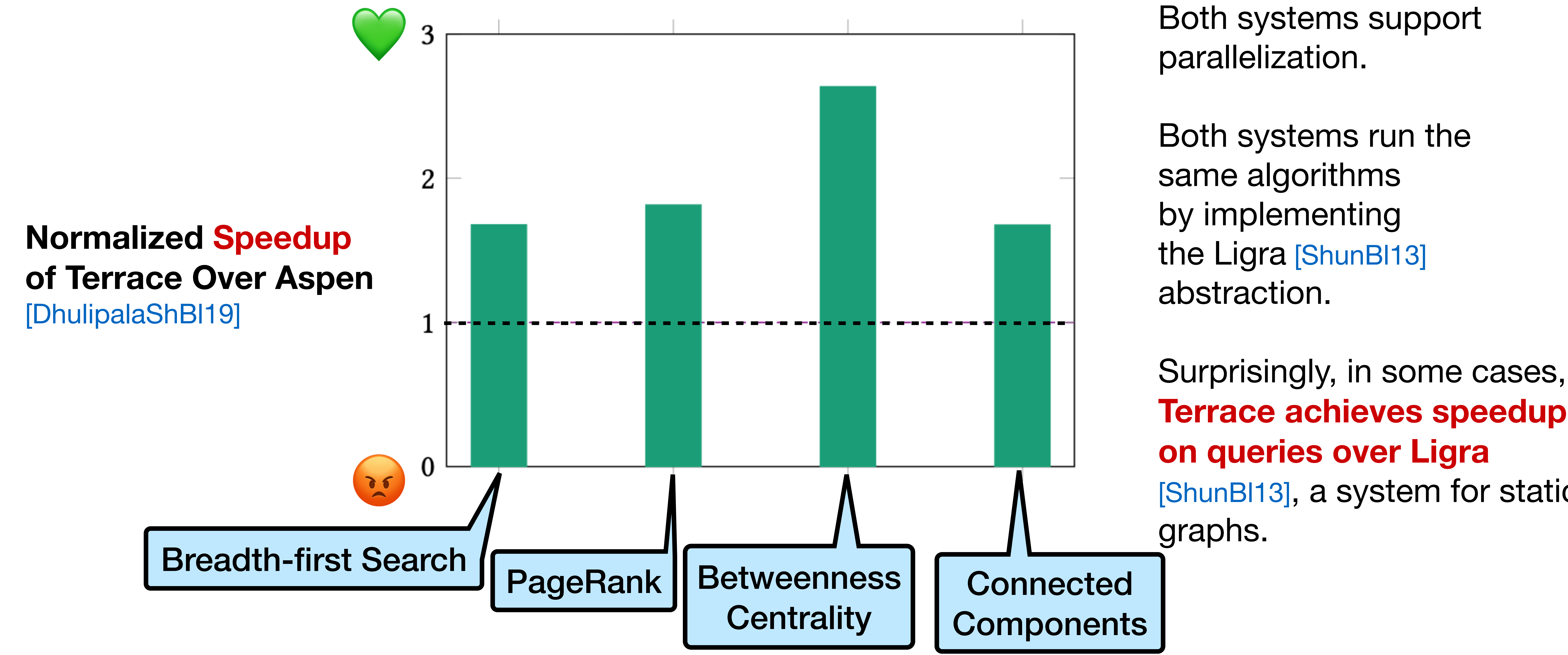



Problem: **Neither data structure clearly wins** for dynamic graphs because graphs require fast updates and scans

Solution: **use both**, depending on degree

# Query Speed in Dynamic-Graph Data Structures

**Terrace,** a dynamic-graph data structure, uses a hierarchical design that takes advantage of graph structure.



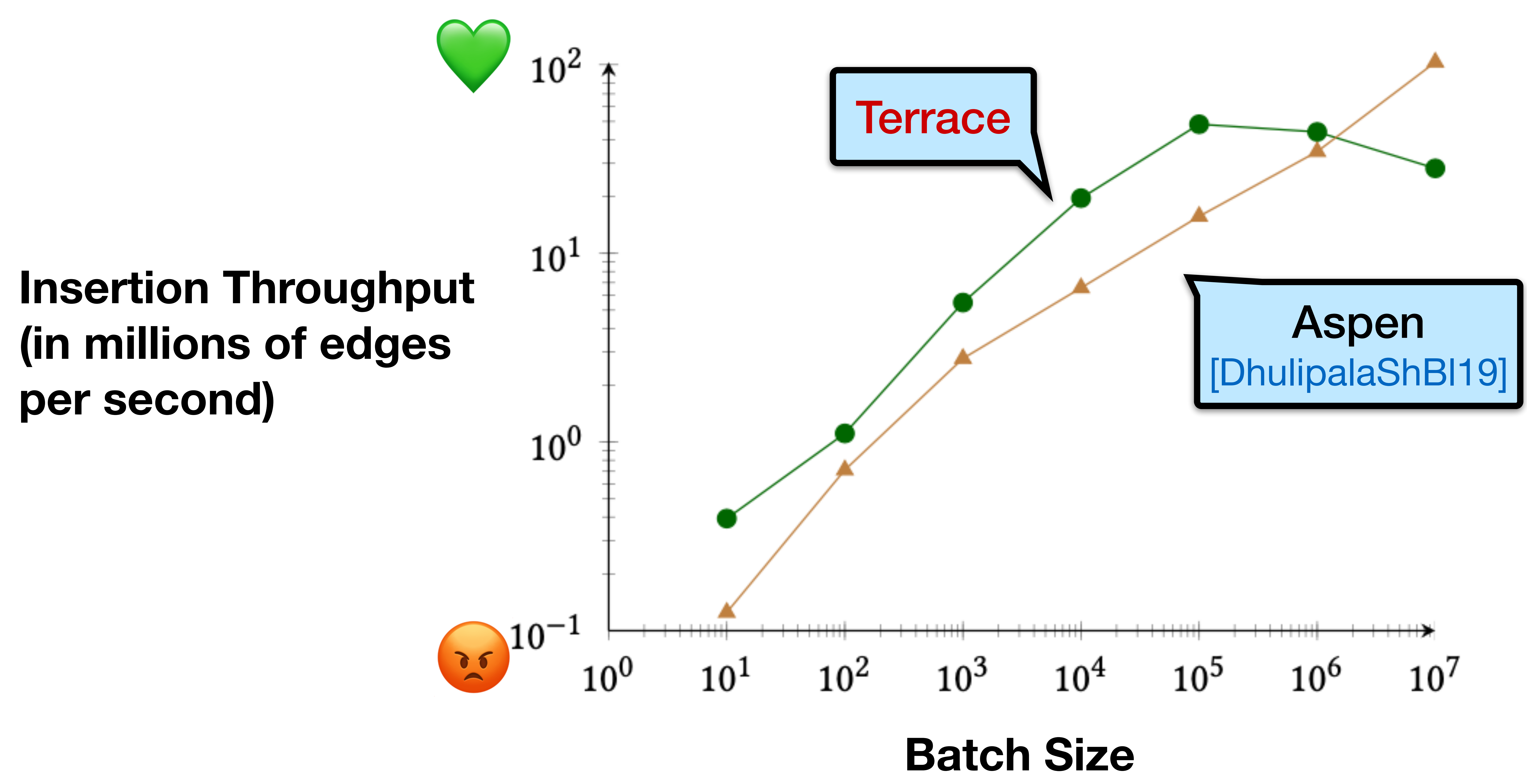

Both systems support parallelization.

Both systems run the same algorithms by implementing the Ligra [ShunBl13] abstraction.

Surprisingly, in some cases, **Terrace achieves speedup on queries over Ligra** [ShunBl13], a system for static graphs.

# Updatability in Dynamic-Graph Data Structures

Terrace achieves the **best of both worlds** in terms of query and update performance by taking advantage of locality.




Edges were generated using an rMAT distribution [ChakrabatiZhFa04] and added in batches using the provided API.

# Exploiting Skewness Improves Cache-Friendliness

The **locality-first design** in Terrace **reduces cache misses** during graph queries.

Additional optimization: store some edges **in-place for extra spatial locality**

On the LiveJournal graph

| Query | Ligra [ShunBl13] (Static) | Aspen [DhulipalaShBl19] (Dynamic) | **Terrace** [PandeyWhXuBu21] |
|---|---|---|---|
| **Breadth-first Search** | 3.5M | 6.3M | 1.1M |
| **PageRank** | 174M | 197M | 128M |

Cache-friendliness translates into **graph query performance**