Dynamic Graphs: Containers, Frameworks, and Benchmarks

Prashant Pandey
Northeastern University
https://prashantpandey.github.io/

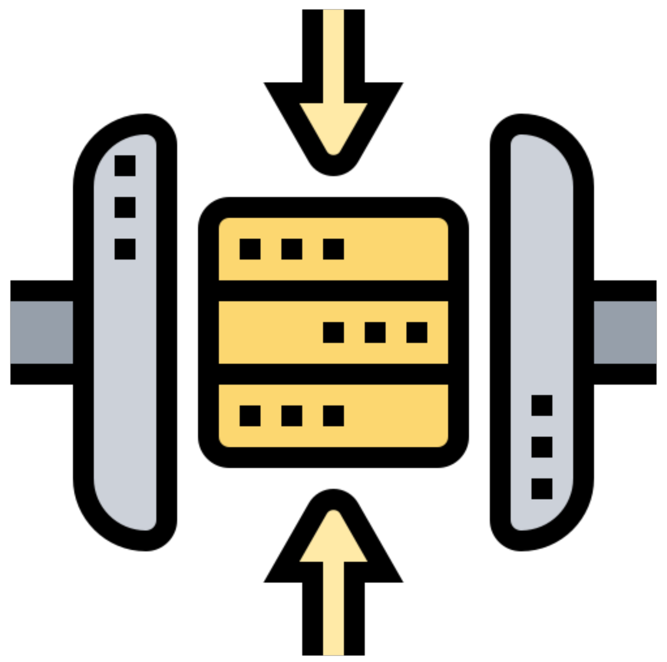
Most slides taken from Prof. Helen Xu, Georgia Tech

My goal as a researcher is to build scalable data systems with strong theoretical guarantees

↓

To scale and *democratize* next-generation data analyses

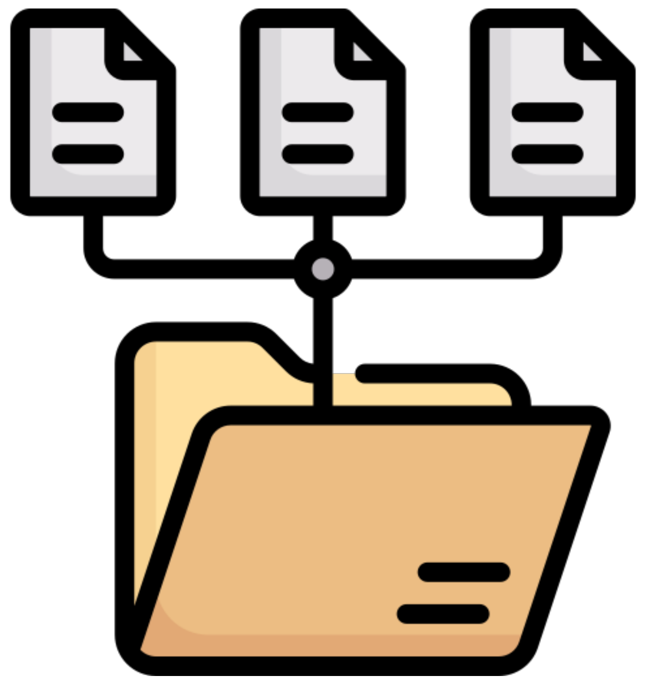
Three approaches to build scalable data systems



Compress it

Goal: make data smaller to fit inside fast memory

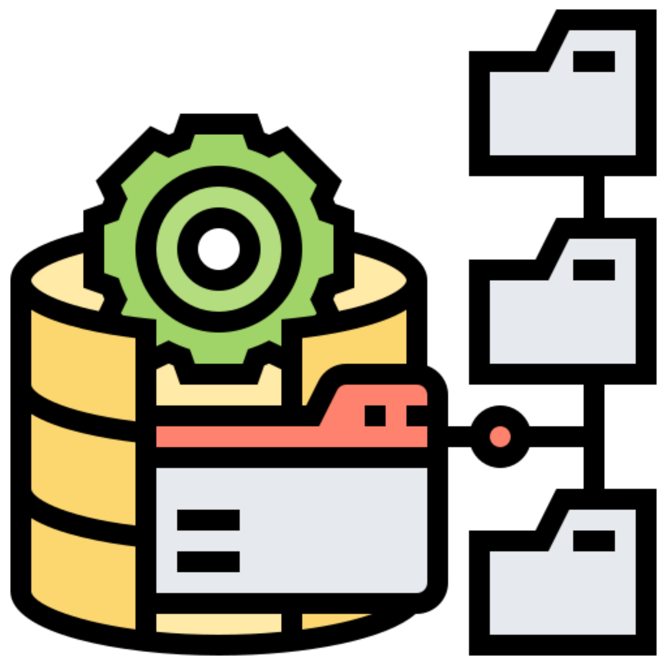
Filters, sketches, succinct data structures



Organize it

Goal: organize data in a I/O friendly way

B-trees, LSM-trees, B^e-trees



Distribute it

Goal: distribute data & reduce inter-node communication

Distributed hash tables

Vertically integrated research

COMPRESS ORGANIZE DISTRIBUTE

Applications

- Squeakr, deBGR, Mantis, Rainbowfish
  BIOINFORMATICS 17,18
  ISMB 17, RECOMB 18, 19
  Cell Systems 18
- LSM-Mantis
  BIOINFORMATICS 22
- VariantStore
  Genome Biology 21
- MetaHipMer*
  IPDPS 21, ACDA 23
- RDM (GNN)
  IPDPS 23

Systems

- B^{e}trFS File System
  FAST 15, 16, SPAA 19
  TOS 16, 17, TOPC 21
- LERT
  (Anomaly detection)
  SIGMOD 20, TODS 21
- Terrace
  (Graph system)
  SIGMOD 21, VLDB 24
- IONIA
  (Distributed KV)
  FAST 24

Data structures

- Quotient filter
  SIGMOD 17, 21, 24
- AdaptiveQF
  SIGMOD 25
- Buffered CMS
  ESA 18, APOCS 23
- OMH
  ISMB 19
- IcebergHT
  SIGMOD 23
- BP-tree
  VLDB 23
- GPU filters & HT
  PPoPP 23, 24
- ZombieHT
  SIGMOD 25

Vertically integrated research

COMPRESS ORGANIZE DISTRIBUTE

Applications

Squeakr, deBGR, Mantis, Rainbowfish
BIOINFORMATICS 17,18
ISMB 17, RECOMB 18, 19
Cell Systems 18

LSM-Mantis
BIOINFORMATICS 22

VariantStore
Genome Biology 21

MetaHipMer*
IPDPS 21, ACDA 23

RDM (GNN)
IPDPS 23

Systems

BetrFS File System
FAST 15, 16, SPAA 19
TOS 16, 17, TOPC 21

LERT
(Anomaly detection)
SIGMOD 20, TODS 21

Terrace, BYO
(Graph systems)
SIGMOD 21, VLDB 24

IONIA
(Distributed KV)
FAST 24

Data structures

Quotient filter
SIGMOD 17, 21, 24

AdaptiveQF
SIGMOD 25

Buffered CMS
ESA 18, APOCS 23

OMH
ISMB 19

IcebergHT
SIGMOD 23

BP-tree
VLDB 23

GPU filters & HT
PPoPP 23, 24

ZombieHT
SIGMOD 25

Terrace: A hierarchical graph container for skewed dynamic graphs

Pandey, Wheatman, Xu, Buluc
SIGMOD ’2021

Survey of Dynamic-graph Data Structures

There has been a long line of work (20+ papers) on developing **dynamic-graph data structures** with fast algorithms and updates. Including (but definitely not limited to):

- Stinger [Ediger, McColl, Riedy, Bader - HPEC ’12]
- Aspen [Dhulipala, Blelloch, Shun - PLDI ’19]
- DGAP [Islam and Dai - SC ’23]

Many of them implement updates as **parallel batches** which insert/delete many elements at the same time [BaderMa07, FriasSi07, BarbuzziMiBiBo10, ErbKoSa14, SunFeBl18, TsengDhBl19, DhulipalaBlSh19, DhulipalaBlGuSu22]:

Batch updates

Introduction to 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)

O(1) to update
O(n) to scan nghs

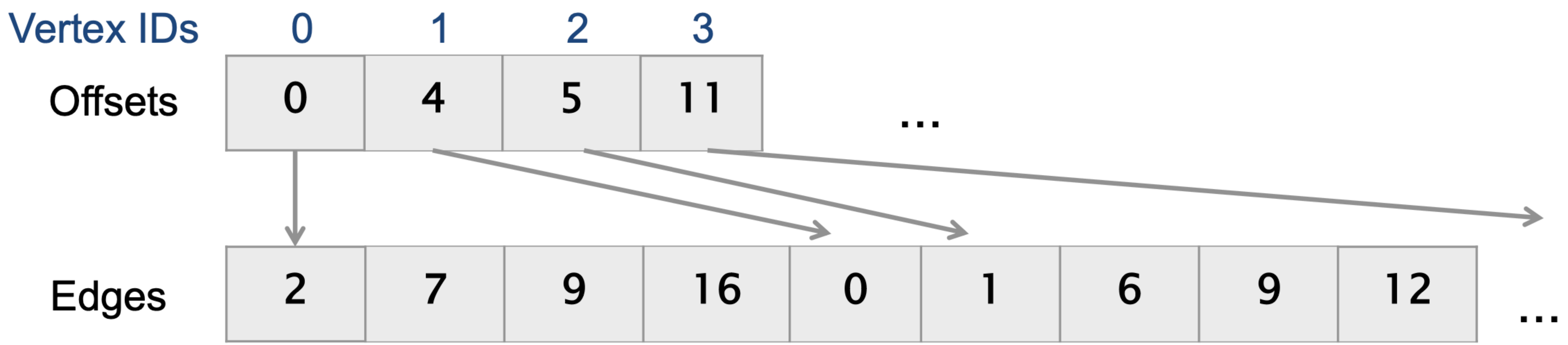
(0,1)
(1,0)
(1,3)
(1,4)
(2,3)
(3,1)
(3,2)
(4,1)

Edge list

Introduction to Graph Representations

In practice, graphs are usually represented in Compressed Sparse Row (CSR) [TinneyWa67] format.

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



CSR is "ideal" for **algorithm performance**, but does not efficiently support **updates**.

O(m) to update
O(deg(v)) to scan nghs of vertex v

Spatial Locality Determines Graph Query Performance

Dynamic-graph data structures (**containers**) 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].

Scan

```
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)
```

Breadth-first search

Scan

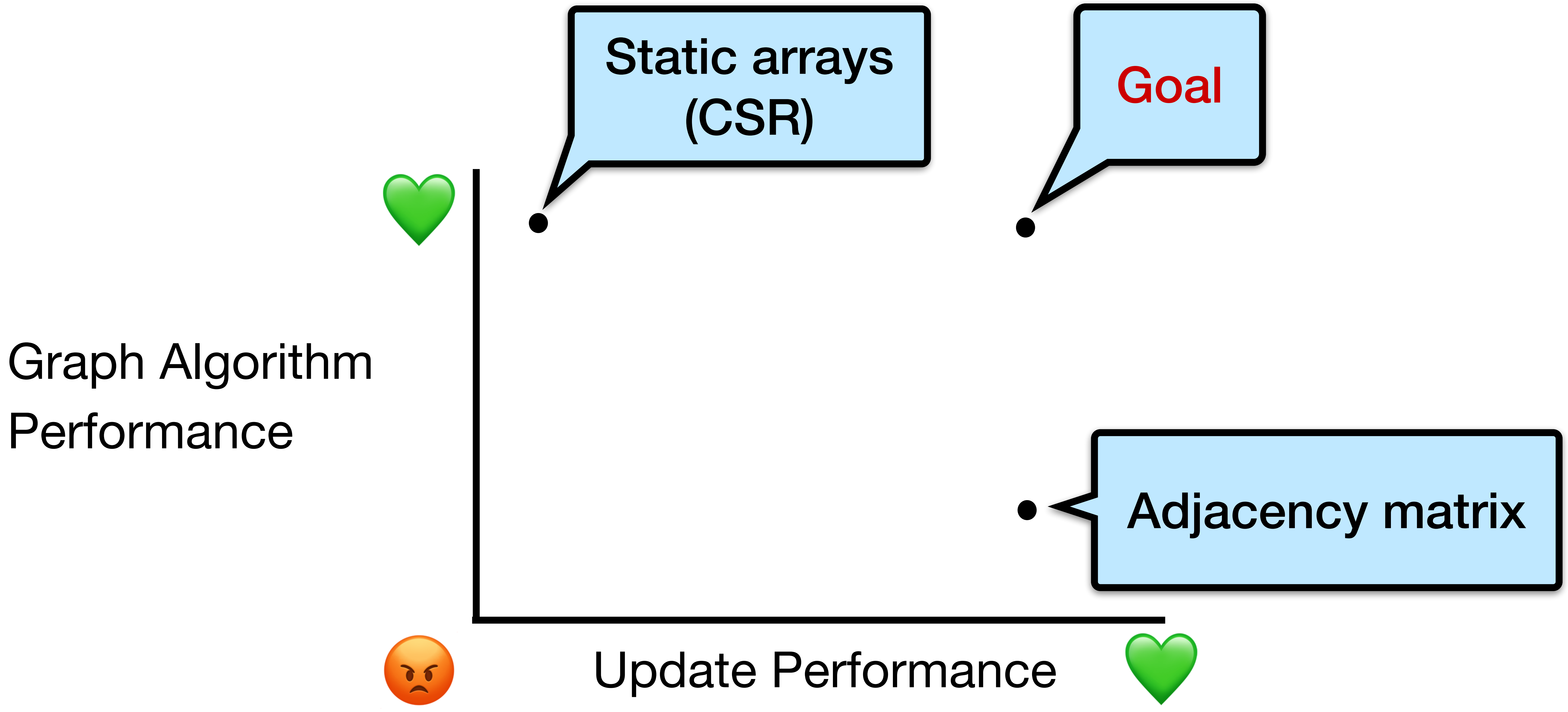
```
Input: graph G
let triangle_count = 0
let E = G.edges()
for (u, v) in E:
  intersect neighbors of u and v:
    if u and v share a neighbor w:
      triangle_count++;
```

Triangle counting

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

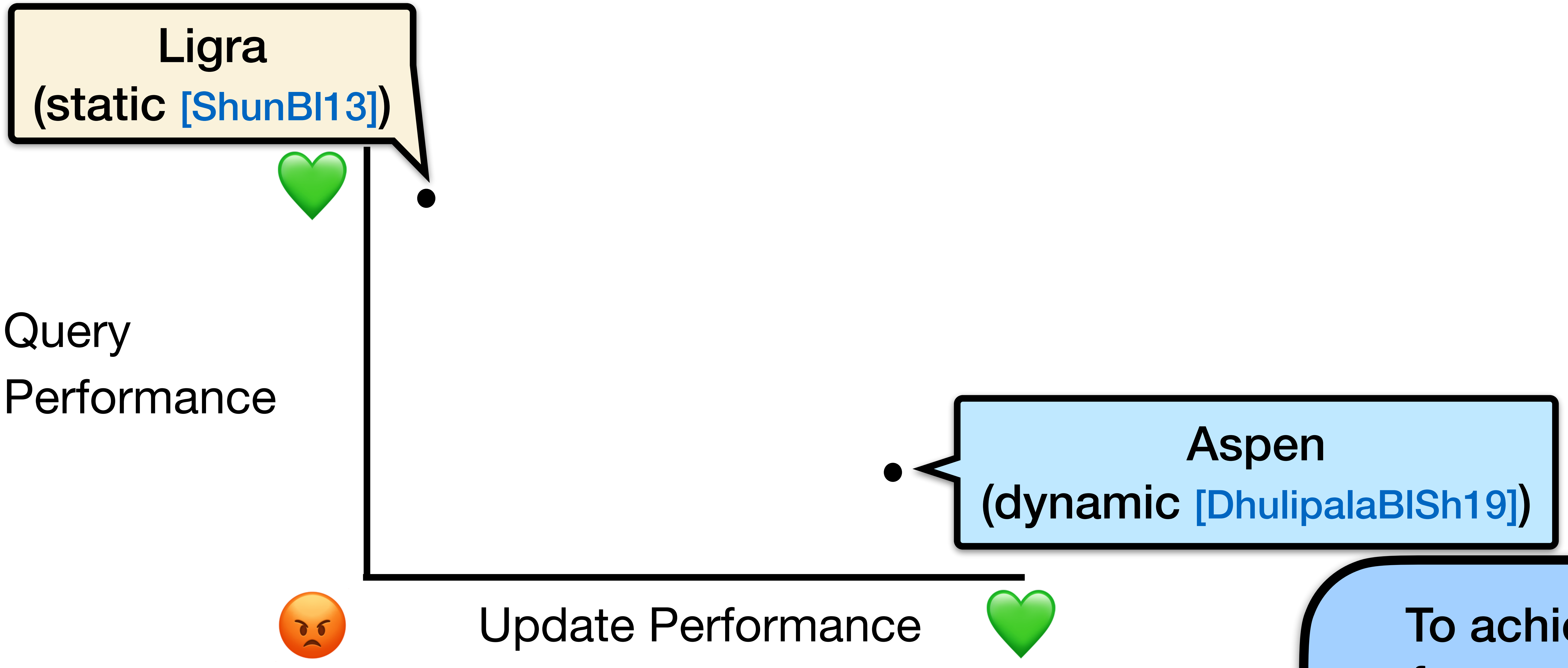
Tradeoff between Locality and Updatability

Problem: Can we choose data structures to support efficient scans and updates for dynamic graphs? i.e., “dynamic CSR”?



Existing Graph Data Structures Trade Off Query and Update Performance

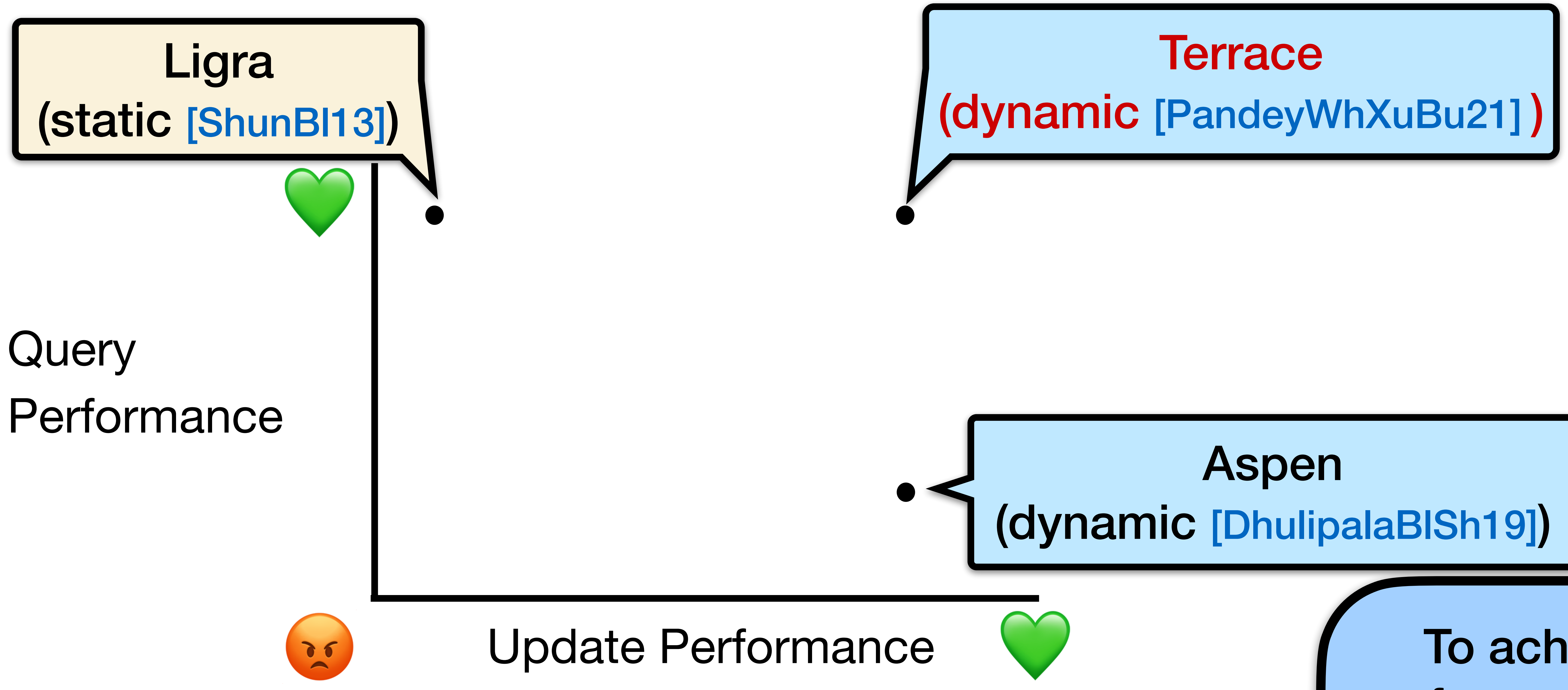
The commonly-held belief about graph data structures says that **query performance trades off with update performance** [EdigerMcRiBa12, KyrolaBlGu12, ShunBl13, MackoMaMaSe15, DhulipalaBlSh19, BusatoGrBoBa18, GreenBa16] due to **data representation** choices.



To achieve good performance, all of the systems are **parallel**.

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

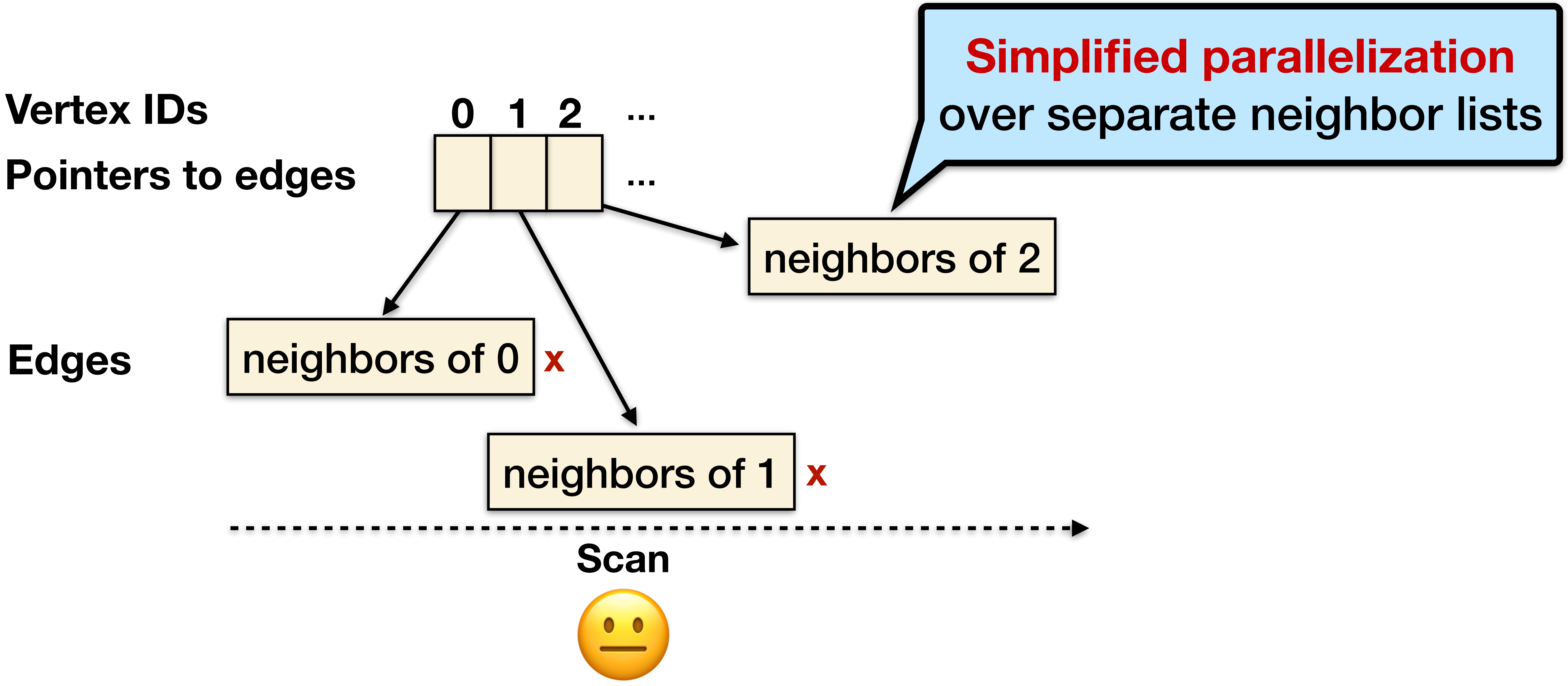
Terrace **achieves good query and update performance** by using data structures that enhance spatial locality.



To achieve good performance, all of the systems are **parallel**.

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

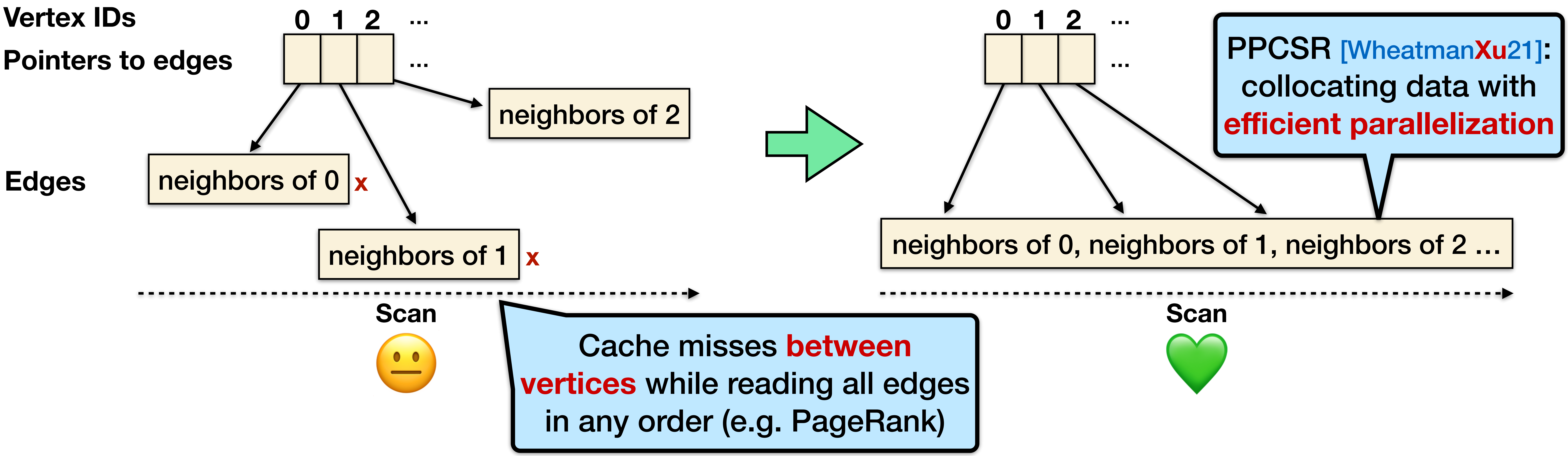
Existing dynamic graph systems optimize for parallelism first with **separate per-vertex data structures** e.g., trees [DhulipalaBlSh19], adjacency lists [EdigerMcRiBa12], and others [KyrolaBlGu12, BusatoGrBoBa18, GreenBa16].



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?

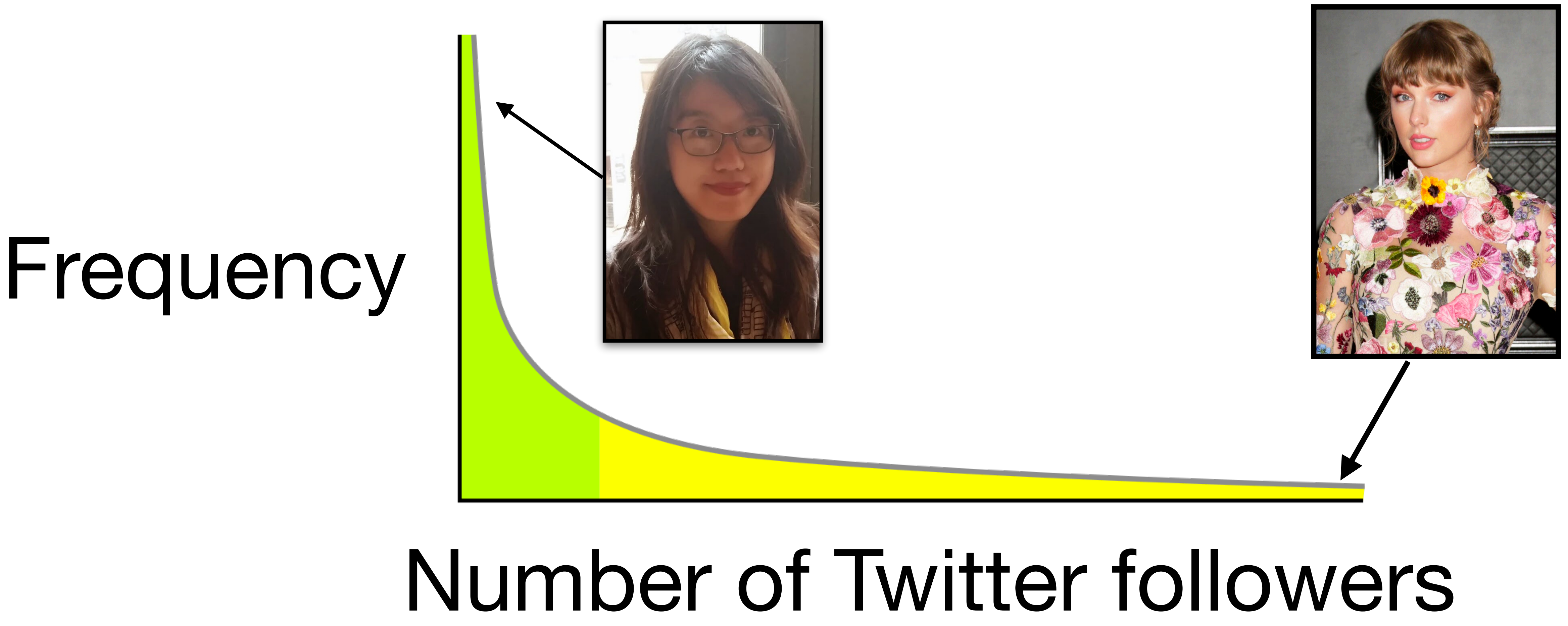
[WheatmanXu21] Wheatman and Xu. "A Parallel Packed Memory Array to Store Dynamic Graphs." ALENEX '21.

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:

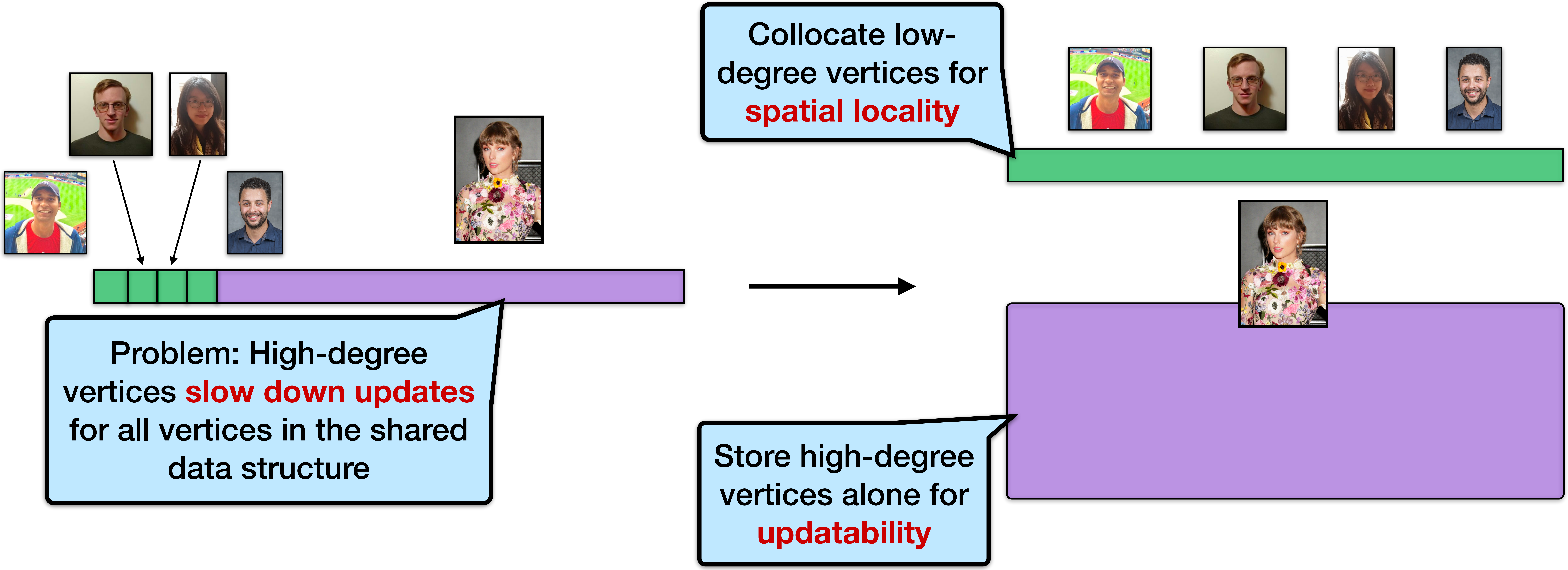
Graph	% < 10 neighbors	% < 1000 neighbors
Twitter [BeamerAsPa15]	64.6	99.5



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

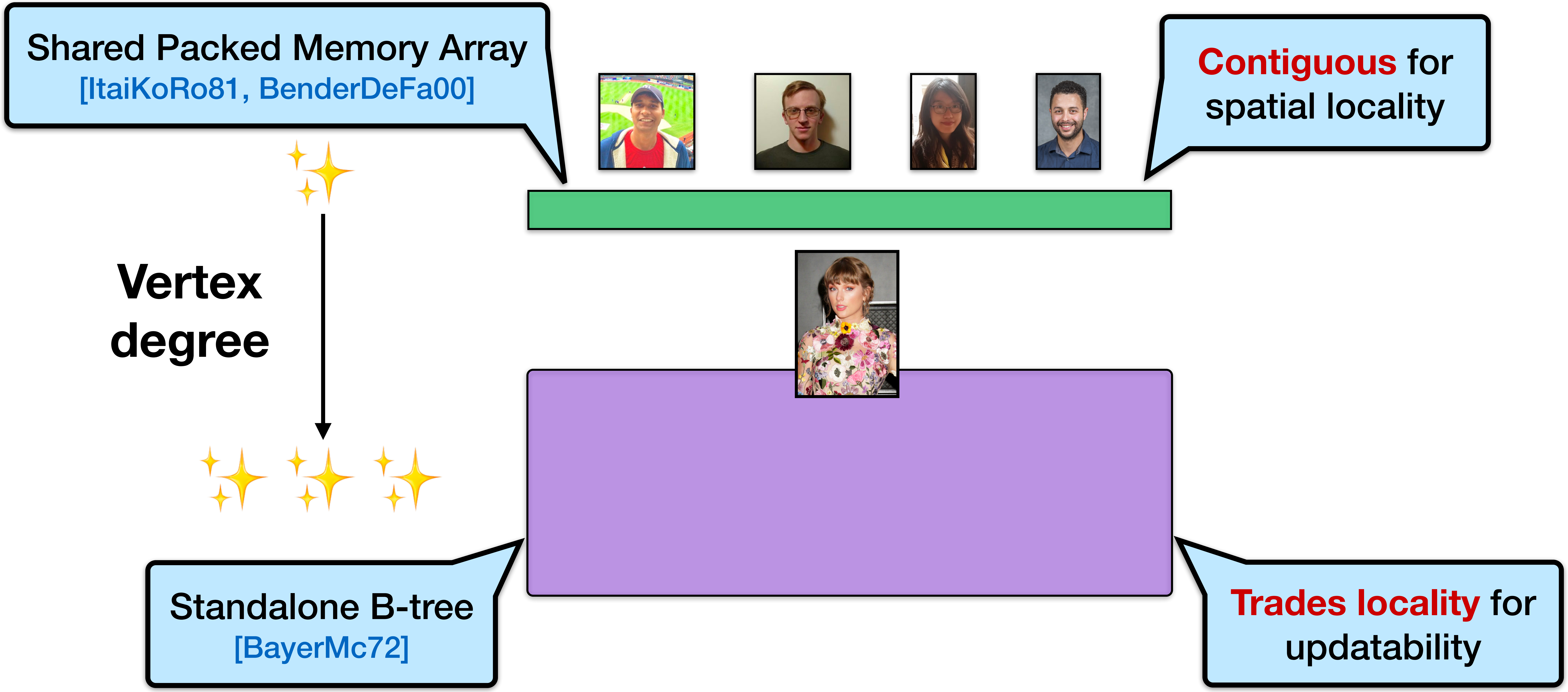
Insight: Further Optimizing for Locality with a Hierarchical Skew-Aware Design

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**.



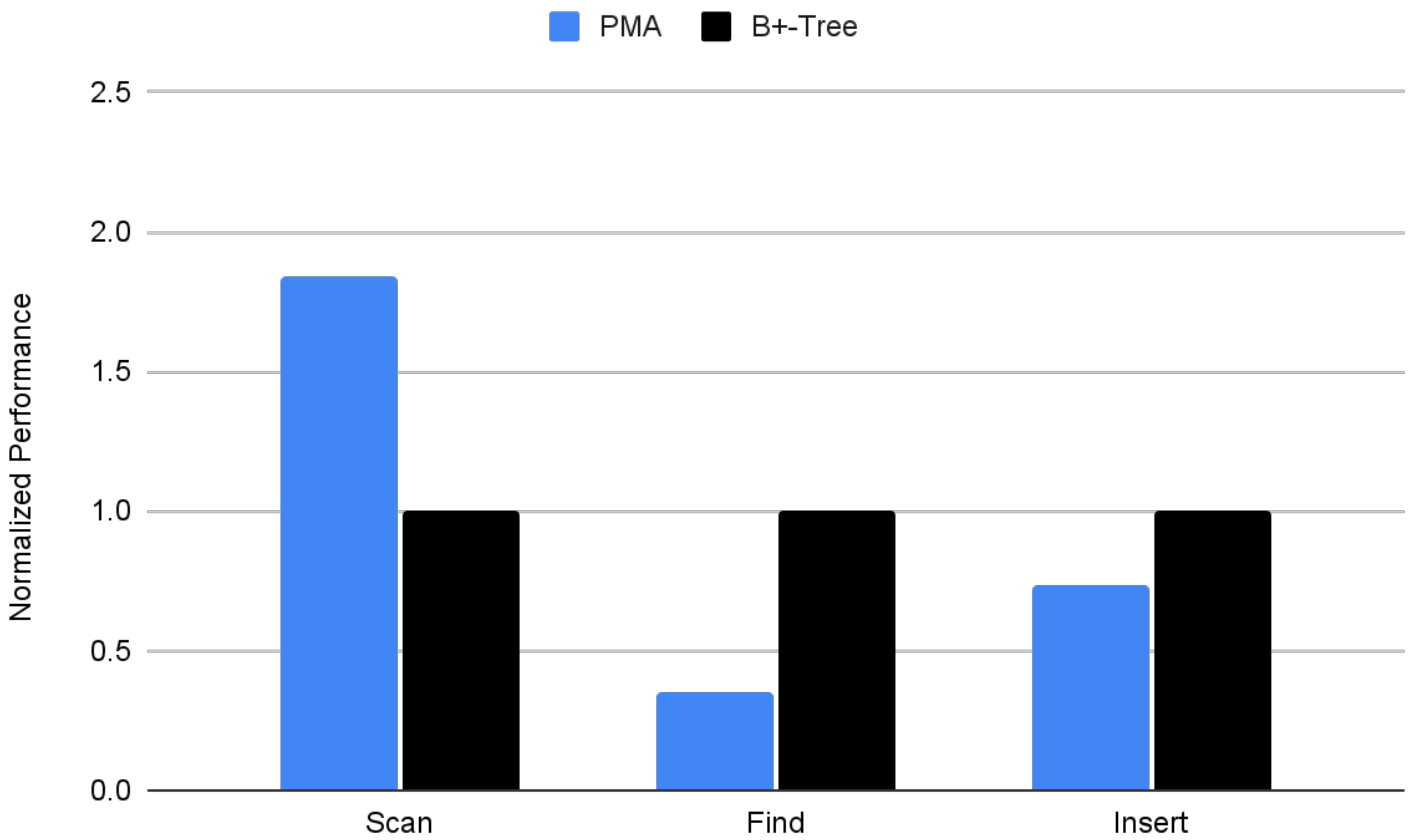
Selecting Data Structures for Dynamic Graphs

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 $O(\log^2(N))$.

The theory does not capture **sequential vs random access**



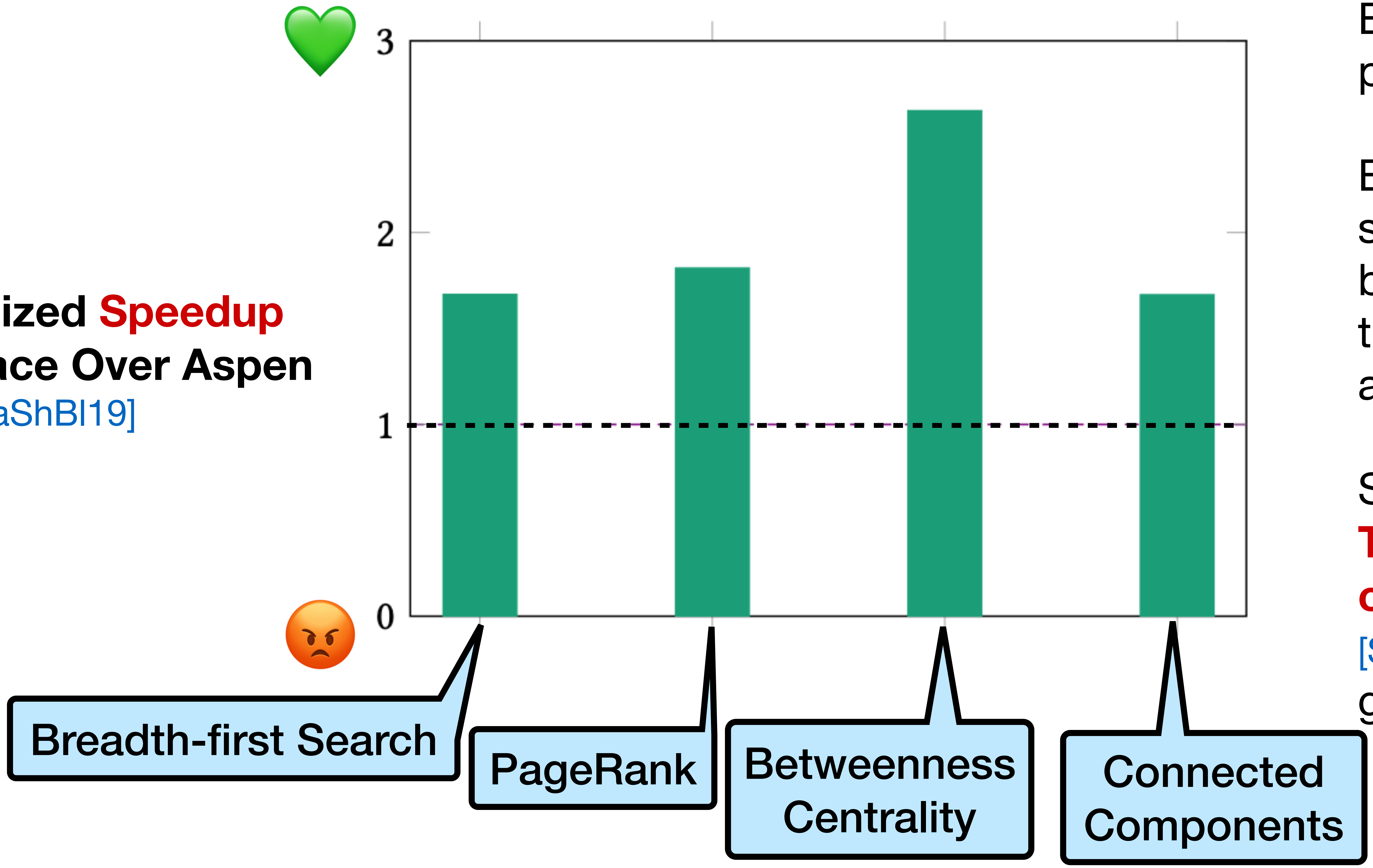
Problem: **Neither data structure clearly wins** for dynamic graphs because graphs require fast updates and scans

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.

Normalized Speedup of Terrace Over Aspen
[DhulipalaShBl19]



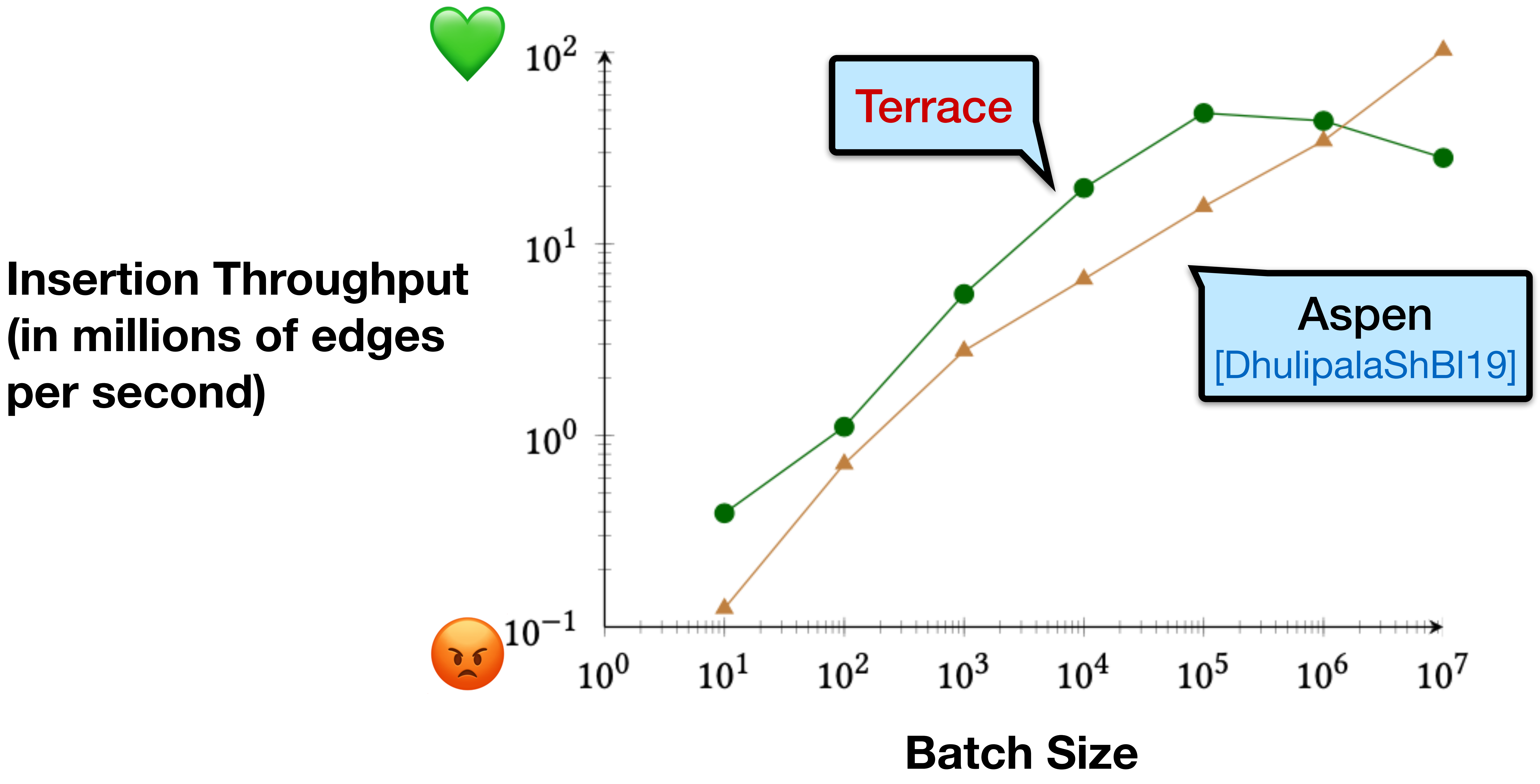
Both systems support parallelization.

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.

Static

Dynamic

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

On the LiveJournal graph

Query	Ligra [ShunBl13]	Aspen [DhulipalaShBl19]	Terrace [PandeyWhXuBu21]
Breadth-first Search	3.5M	6.3M	1.1M
PageRank	174M	197M	128M

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

Fair and Comprehensive Benchmarking of Dynamic-Graph Containers

from “BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers,”
Wheatman, Dong, Shen, Dhulipala, Łącki, **Pandey**, and Xu.
VLDB ’24

Results Highlights

- The Terrace paper [PandeyWhXuBu21] reports a 1.7-2.6x speedup over Aspen [DhulipalaBlSh19]

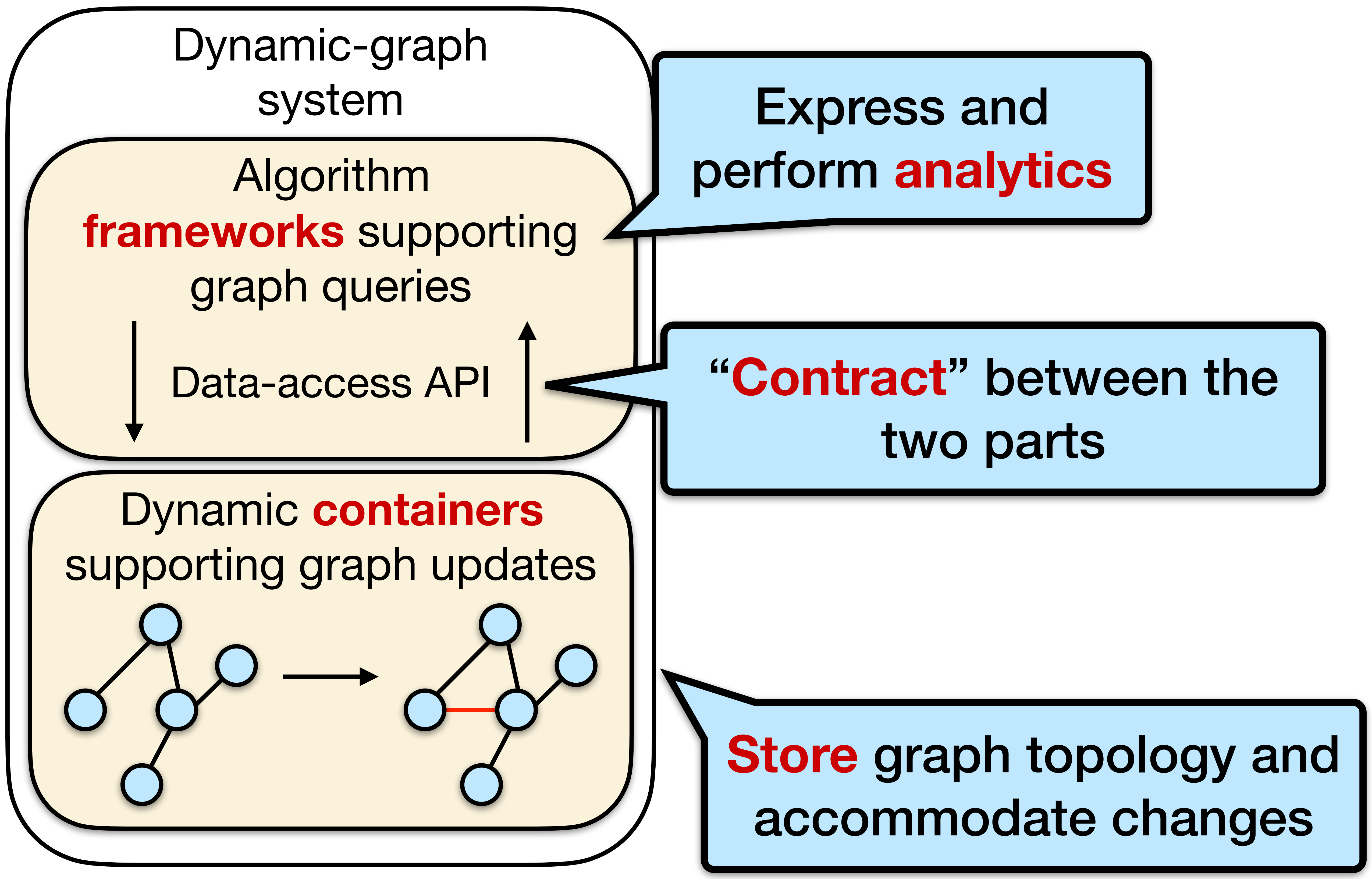
- The Aspen paper [DhulipalaBlSh19] reports 1.8-15x speedup over other dynamic-graph data structures.

- The VCSR paper [IslamDaiCh22] reports speedups of 1.2x-2x speedup over PCSR [WheatmanXu18].

- …other papers report similar ratios

Graph Containers In Dynamic-Graph Systems

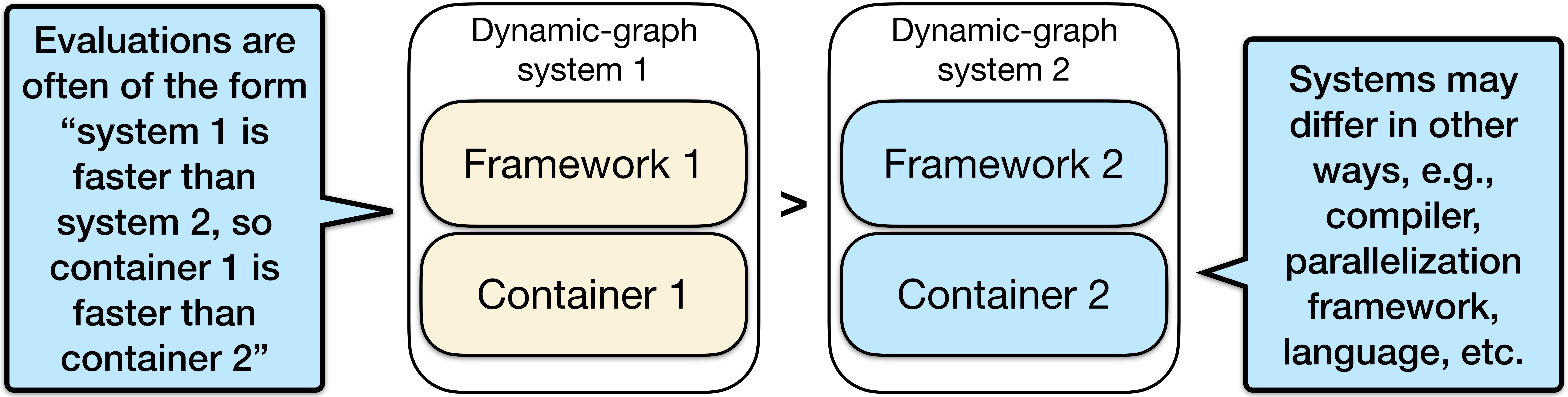
A fundamental design decision in the process of developing any dynamic-graph algorithm is the **choice of the graph container** (i.e., the data structure that represents the graph) [LeoBoncz21, DhulipalaBlGuSu22, DhulipalaBlSh19, EdigerMcRiBa12, …and many others].



Existing Evaluations Compare Overall Systems

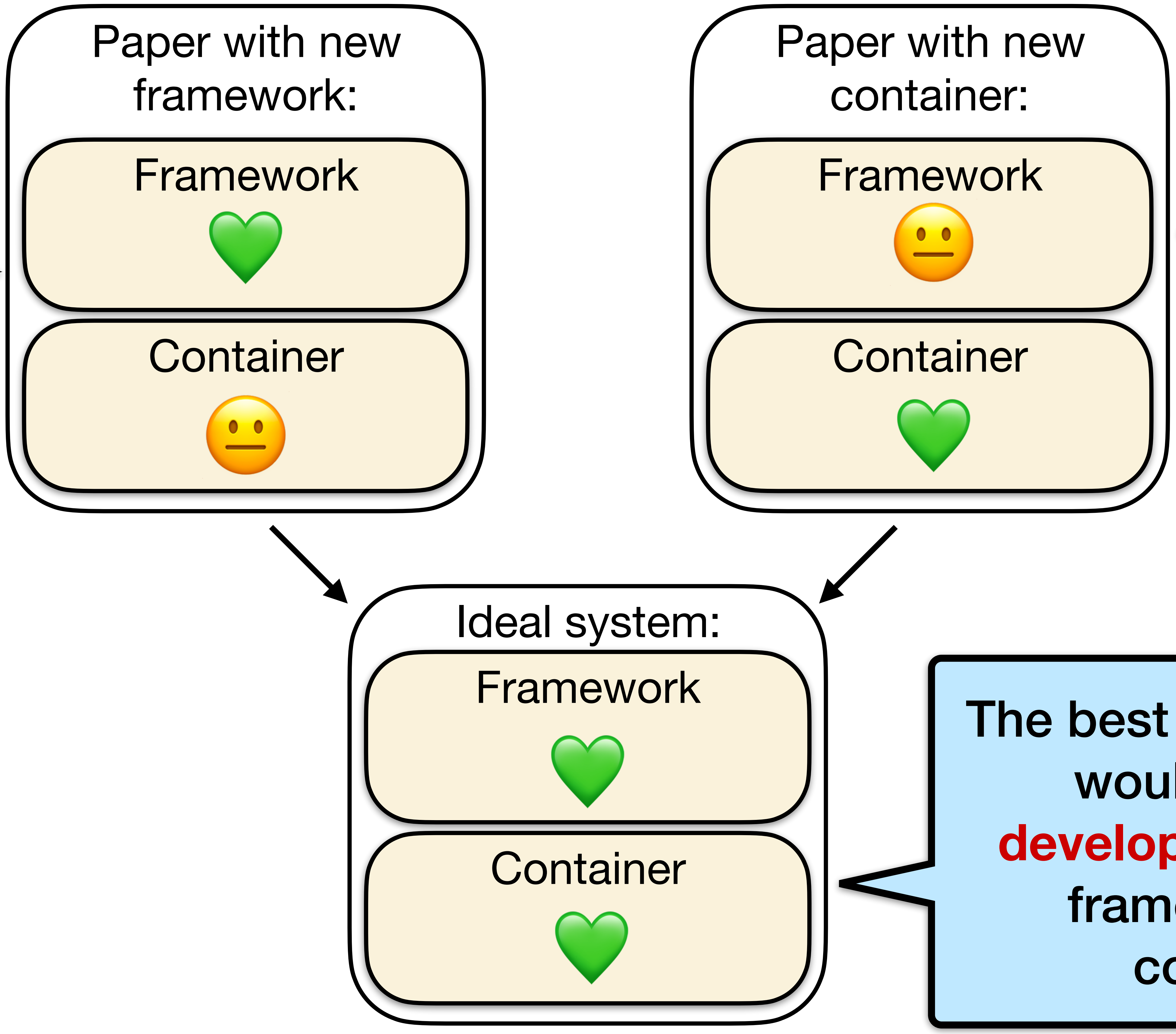
At present, it is almost impossible to answer the question: “which is the **right graph container** for a given application?”

The main reason is because most (if not all) works introducing new dynamic-graph containers perform **end-to-end comparisons** with overall systems as the components are **tightly coupled** in the implementations.



Existing Systems Usually Optimize One Component Only

Framework developers often implement an **ad hoc container** with limited functionality or performance

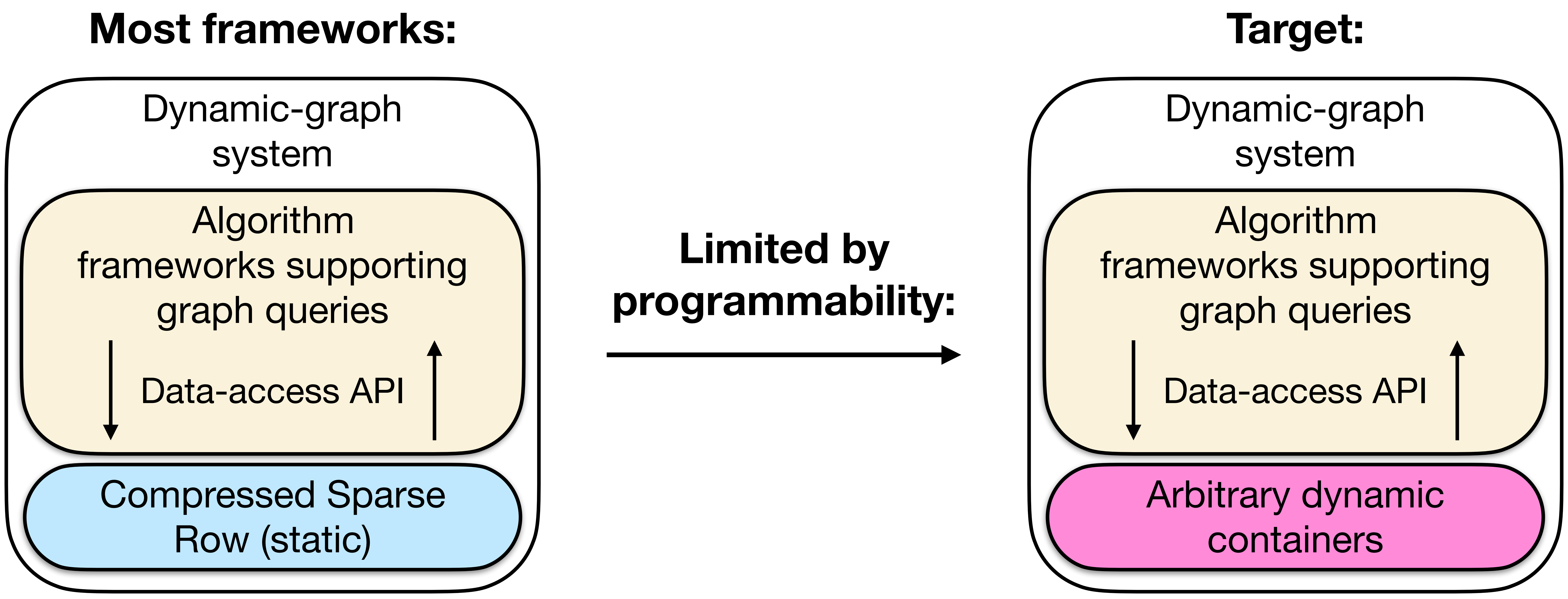


Container developers often implement a **minimal framework or use direct implementations** with limited functionality and generality

The best overall systems would **combine developments** in both frameworks and containers

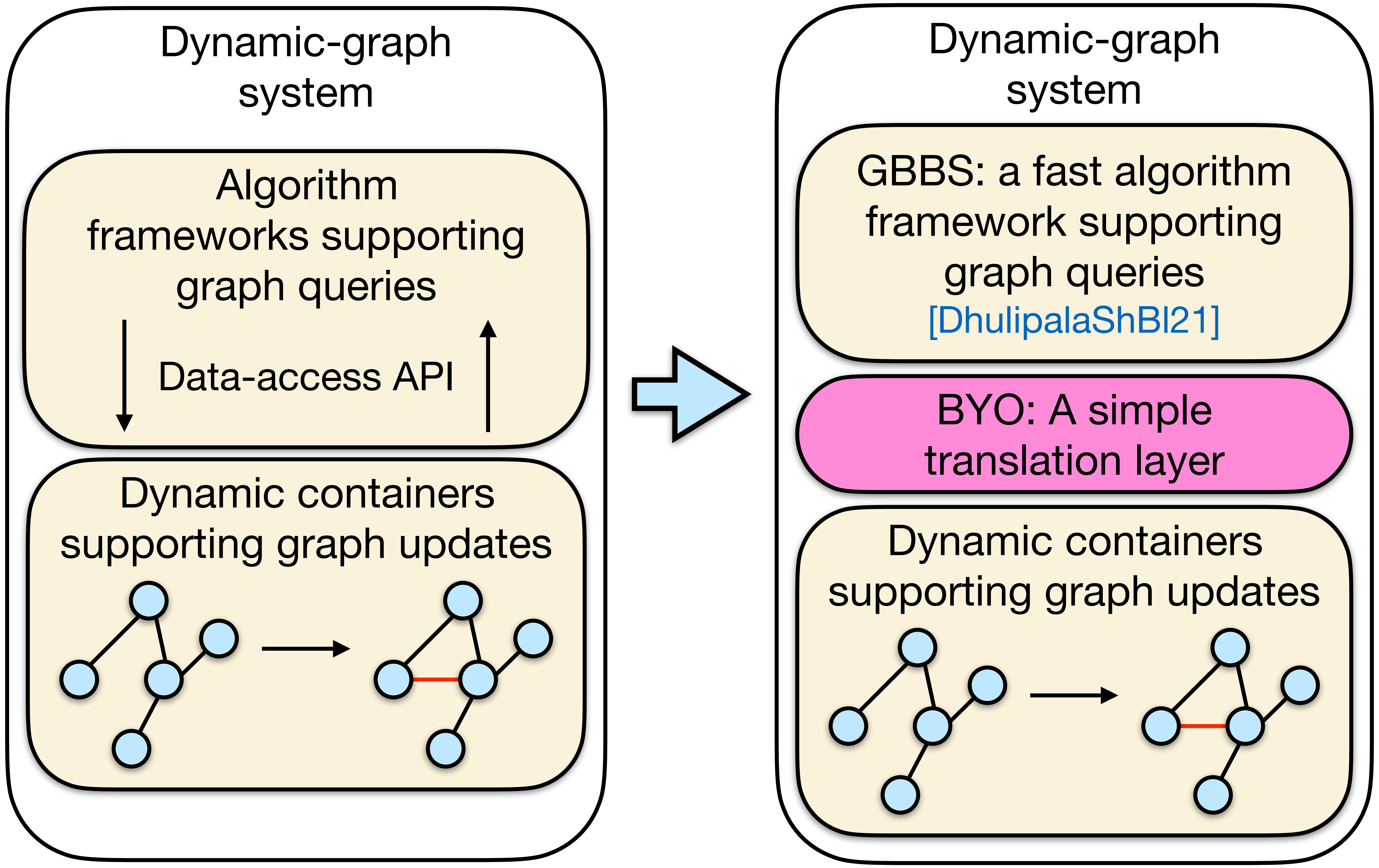
Changing the Container is Challenging in Current Framework Implementations

Graph-algorithm frameworks/standards (e.g., Ligra [ShunBl13], GraphBLAS [Davis23, etc], GBBS [DhulipalaShBl21], etc.) offer hope for standardizing comparisons between containers with high-performance frameworks, but **current implementations are too complex** to easily adapt.



BYO: Simplifying the Intermediate API

We introduce BYO, a simple, easy-to-use **translation layer** between the Graph Based Benchmark Suite [DhulipalaShBl21] and arbitrary graph data structures.



Using BYO, we evaluated **27 different graph containers** (both off-the-shelf and specialized) on a suite of **10 algorithms x 10 graphs**.

Results Highlights

Previously: 🙌

- The Terrace paper [PandeyWhXuBu21] reports a 1.7-2.6x speedup over Aspen [DhulipalaBlSh19]
- The Aspen paper [DhulipalaBlSh19] reports 1.8-15x speedup over other dynamic-graph data structures.
- The VCSR paper [IslamDaiCh22] reports speedups of 1.2x-2x speedup over PCSR [WheatmanXu18].
- …other papers report similar ratios

With standardized evaluation under BYO:

- **All specialized containers** (e.g., Aspen, DHB [GrintenPeWi22], Terrace [PandeyWhXuBu21], etc.) **are within 10% of each other (on average).**
- An **off-the-shelf B+-tree** (from Abseil) is **1.22x** slower than CSR (on average). The fastest **specialized dynamic container** (CPAM [DhulipalaBlGuSu22]) we tested was **1.11x** slower than CSR (on average).

🤏

Results Highlights

Previously: 🙌

- The Terrace paper [PandeyWhXuBu21] reports a 1.7-2.6x speedup over Aspen [DhulipalaBlSh19]
- Tl re her dy
- Tl ts speedups of 1.2x-2x speedup over PCSR [WheatmanXu18].
- …other papers report similar ratios

What does this mean for dynamic-graph data structure developers?

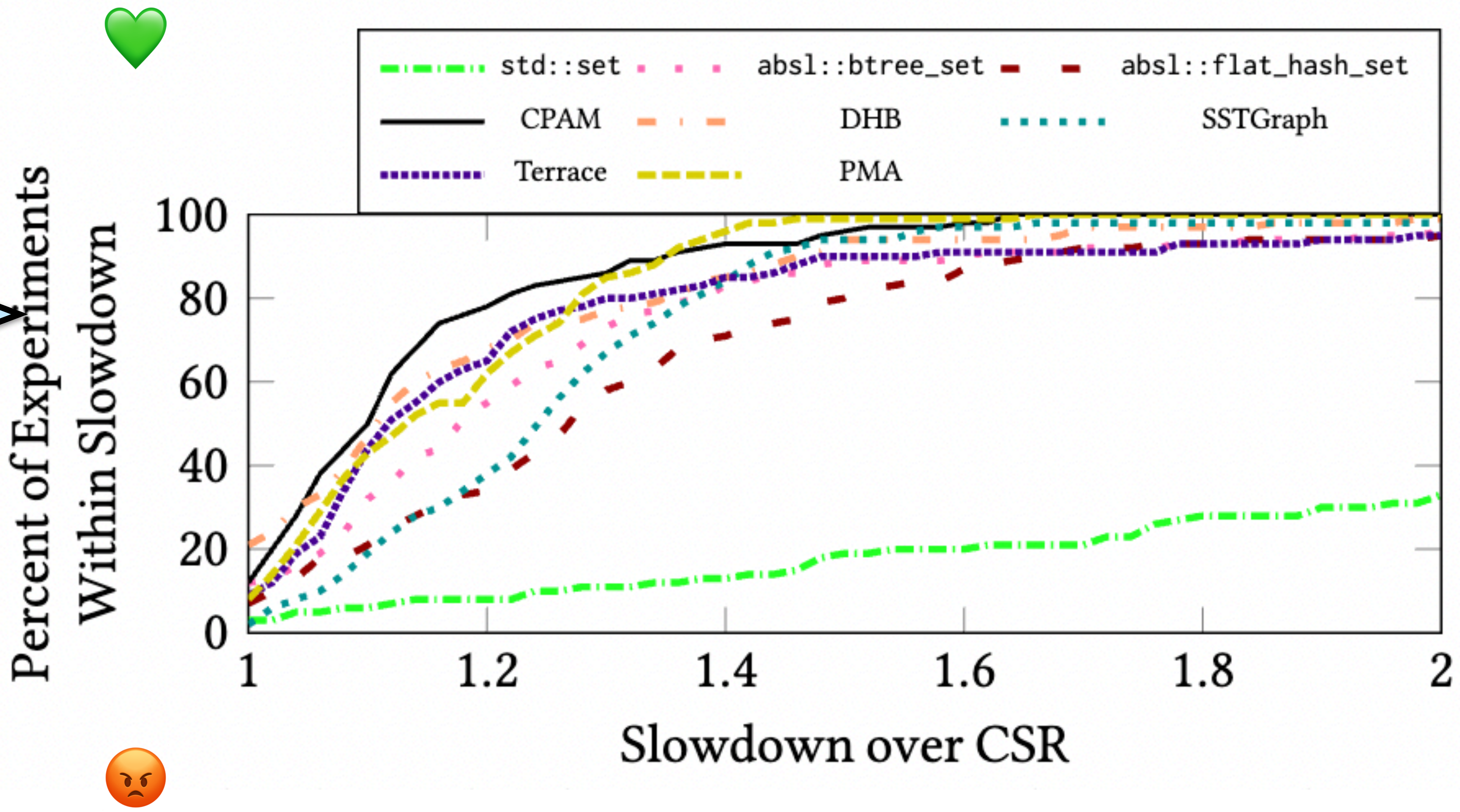
With standardized evaluation under BYO:

- **All specialized containers** (e.g., Aspen, DHB [GrintenPeWi22], Terrace [PandeyWhXuBu21], etc.) **are within 10% of each other (on average).**
- An **off-the-shelf B+-tree** (from Abseil) is **1.22x** slower than CSR (on average). The fastest **specialized dynamic container** (CPAM [DhulipalaBlGuSu22]) we tested was **1.11x** slower than CSR (on average). 🤏

Beyond high-level algorithm performance

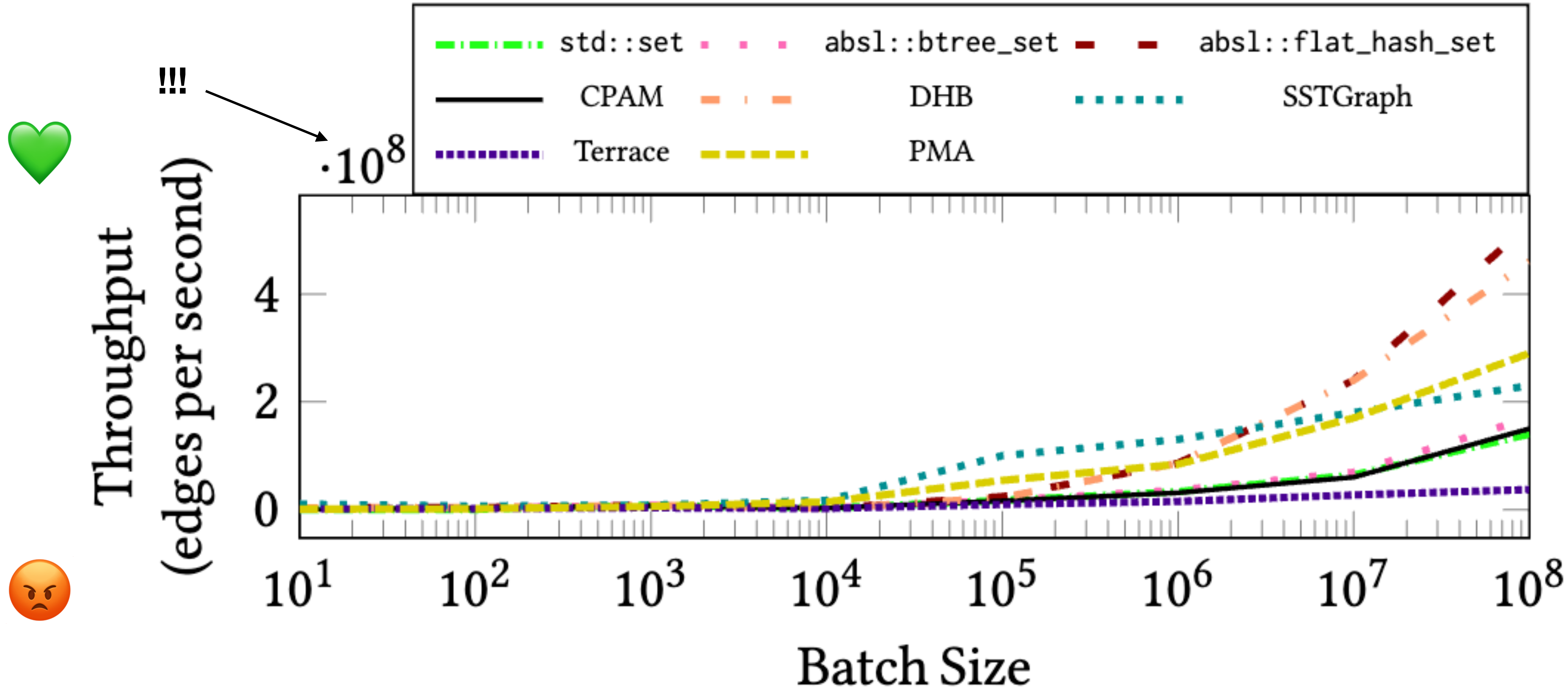
Specialized data structures can improve the worst-case performance on **hard problem instances**.

10 graphs (from 57M to 4B edges) x 10 algorithms = 100 experiments per point

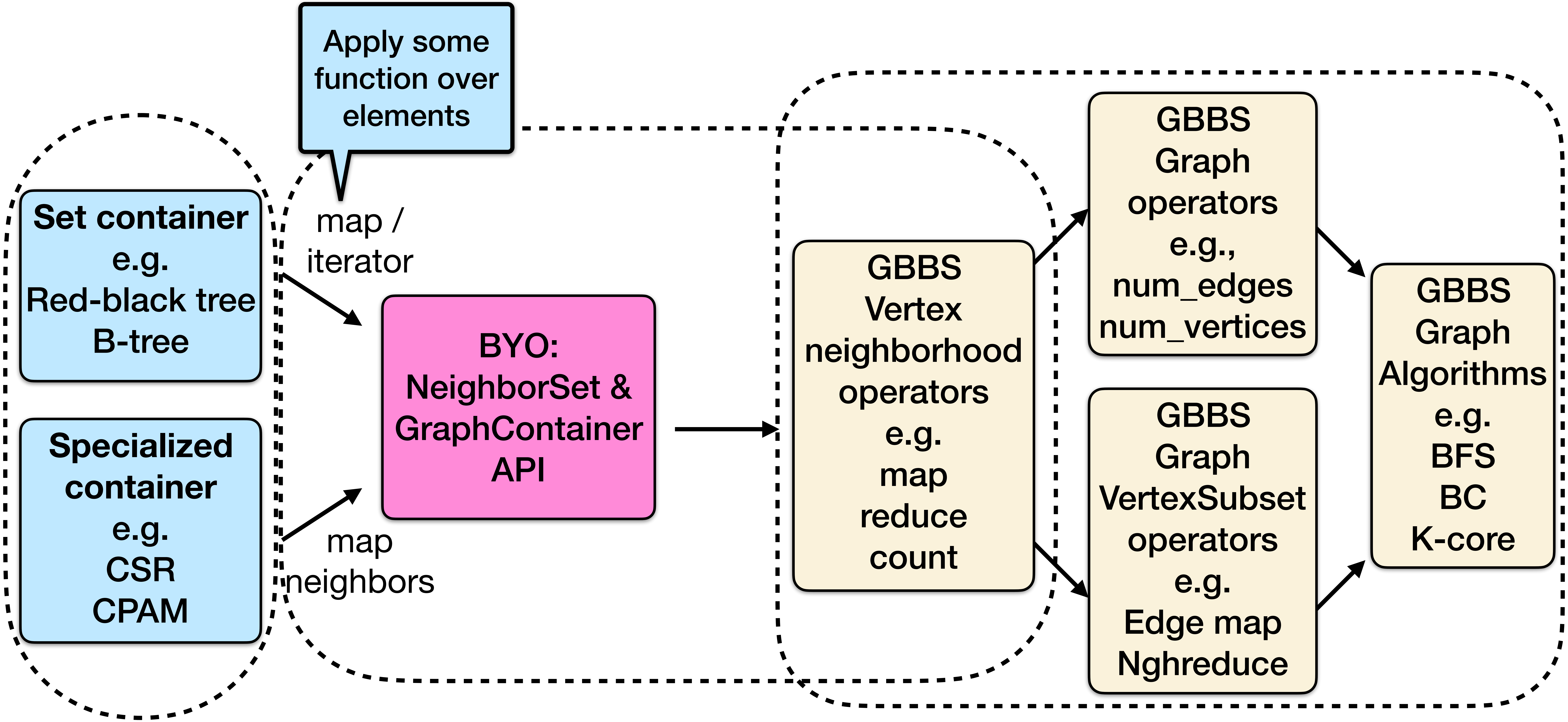


Another axis - update performance

Specialized data structures can also **overcome the classical query-update tradeoff** with parallelization of the update algorithm.

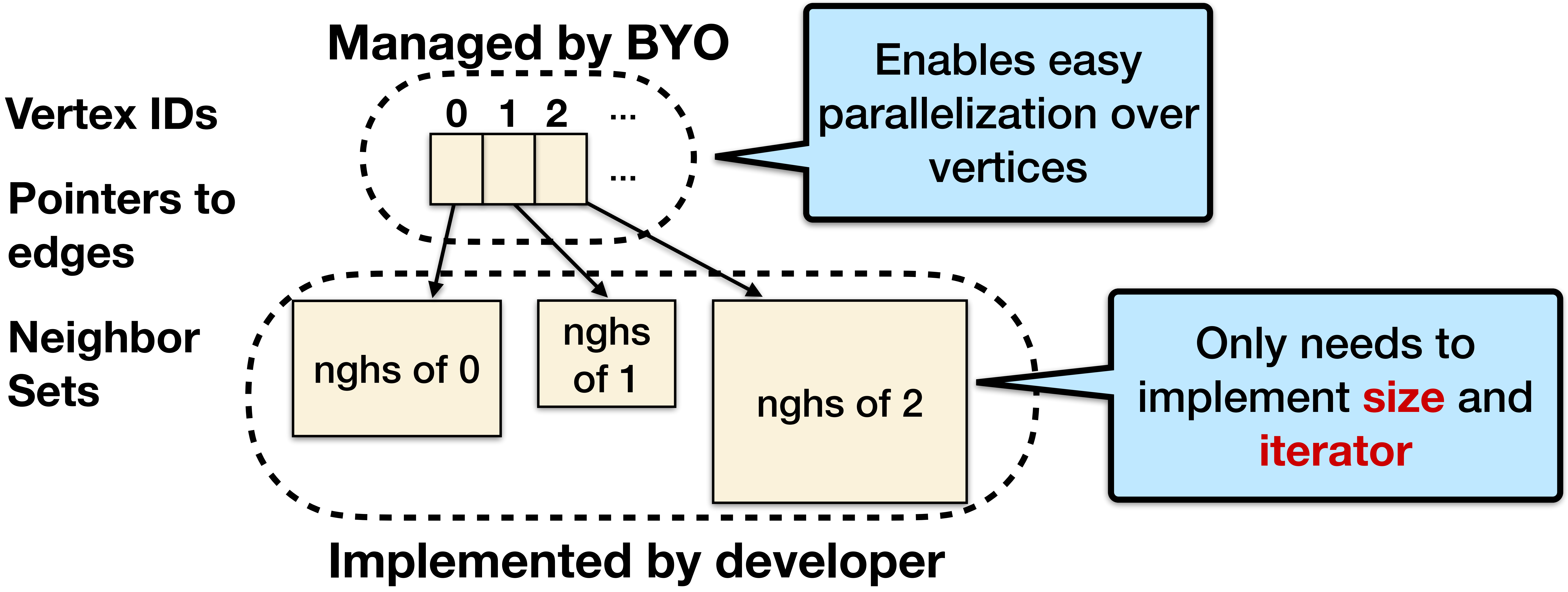


Relationship of System Components and BYO



Connecting BYO to Graph Containers using the NeighborSet API

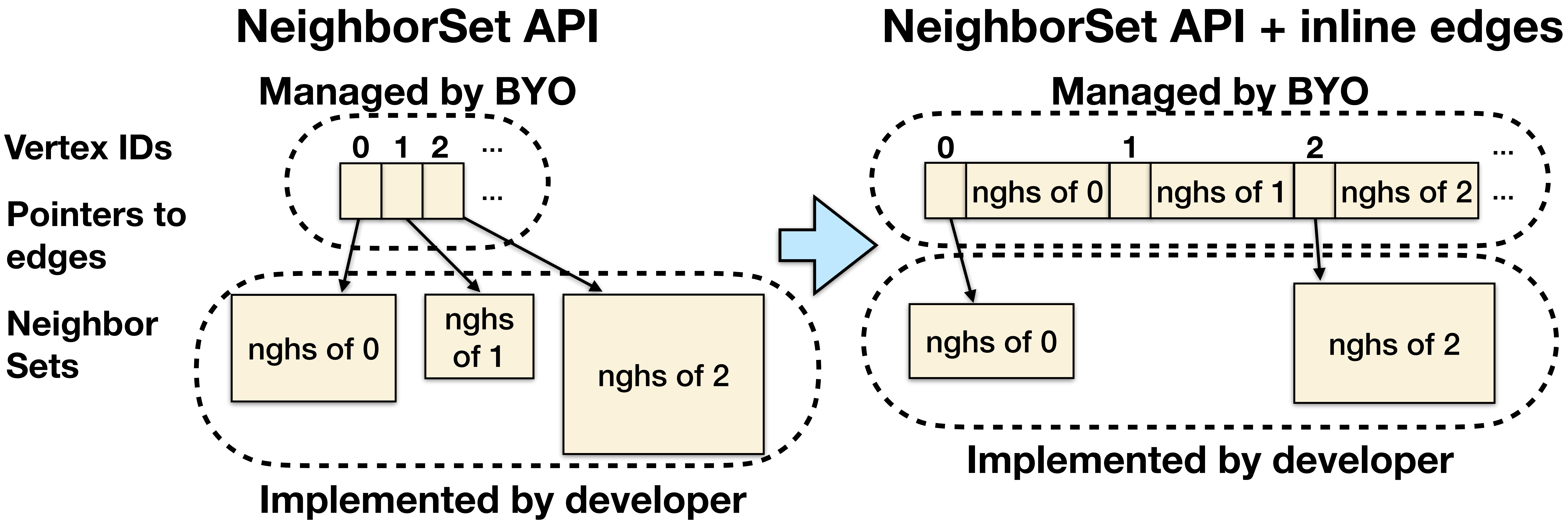
BYO exposes the **NeighborSet abstraction** to capture the two-level sequence-of-sets graph format, which appears in many representations including Stinger [EdigerMcRiBa12] (adjacency lists), Aspen [DhulipalaBlSh19], and CPAM [DhulipalaBlGuSu22] (trees of trees).



Advantages of the NeighborSet API

The NeighborSet API is designed to make it **as easy as possible** for the developer to integrate their container with BYO. It supports **free translation** from the standard C++ STL API.

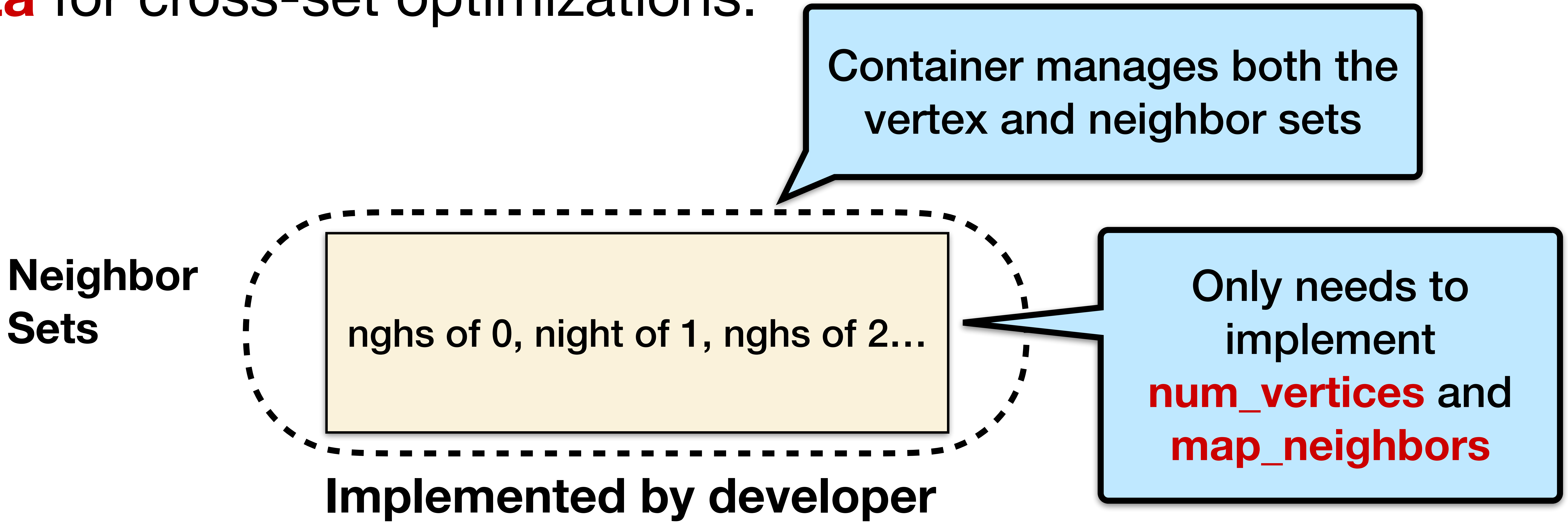
It also incorporates the **inline optimization** from Terrace [PandeyWhXuBu21] to enable overall faster systems.



Connecting BYO to Graph Containers using the GraphContainer API

Some graph containers do not represent neighbor sets as separate independent data structures (e.g., Compressed Sparse Row [TinneyWa67], F-Graph [WheatmanBuBuXu24], Terrace [PandeyWhXuBu21], SSTGraph [WheatmanBu21]).

The GraphContainer API supports these types of containers, which **collocate data** for cross-set optimizations.



All You Need is Map

BYO **simplifies the original GBBS** neighborhood operators such as reduce, count, degree, etc. by implementing several of them with **map**.

Functional primitive that applies an arbitrary **function over a collection** of elements.

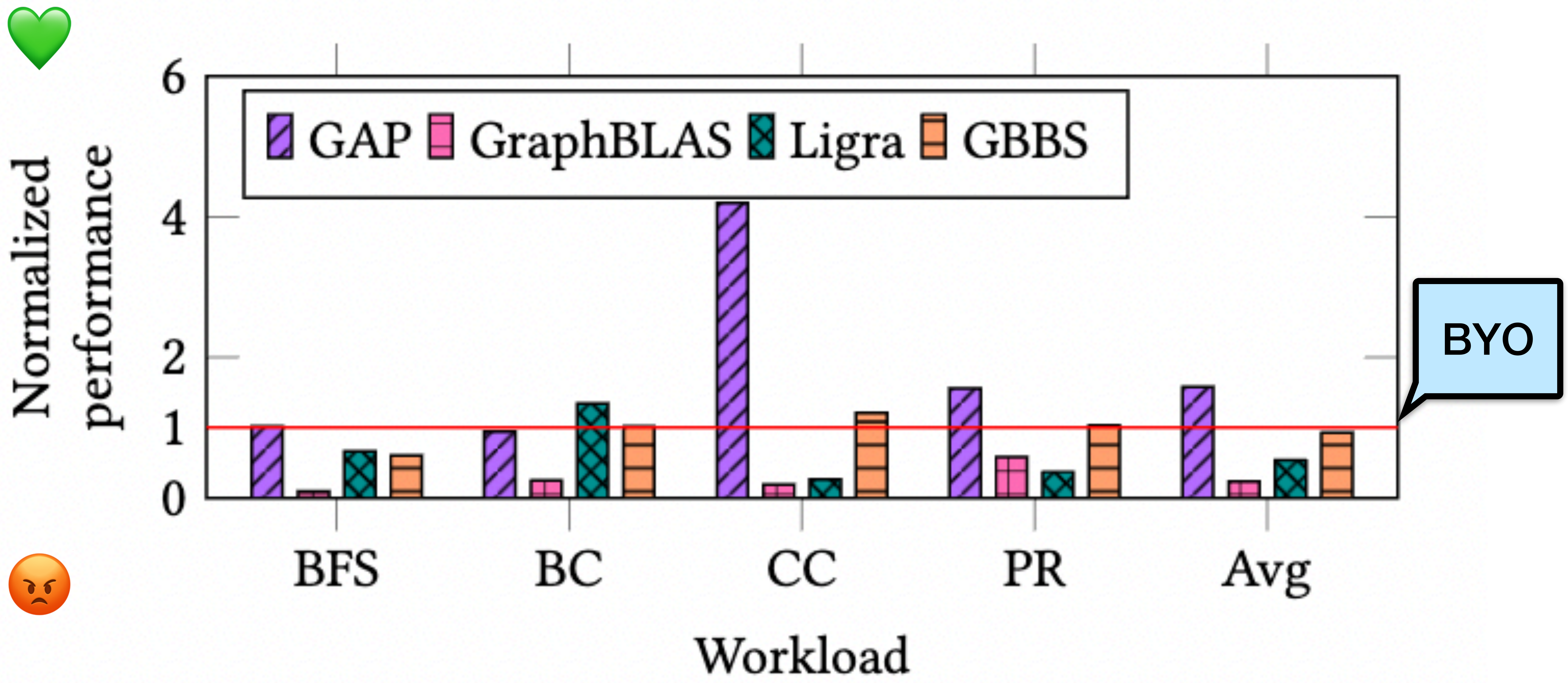
GBBS Vertex Operator	*B.Y.O. Lambda*
Map	Pass through provided function
Reduce	auto value = identity map([&](auto ...args) { value.combine(f(args...)) })
Count	int cnt = 0 map([&](auto ...args) { cnt += f(args...) })
Degree	int cnt = 0 map([&](auto ...args) { cnt += 1) })
getNeighbors	Set ngh = {} map([&](auto ...args) { ngh.add(args) })
Filter	Set ngh = {} map([&](auto ...args) { if (pred(args) ngh.add(args) })

Table 1: GBBS primitives implemented using just the map primitive.

BYO also connects with data structures that implement **more optimized versions of map** (i.e., parallel and early exit).

Empirical Frameworks Comparison

BYO achieves **competitive performance with other frameworks** (Ligra [ShunBl13], GraphBLAS [Davis19, 23], and the GBBS (that it is based on).



BYO Expands the Scope of Algorithms for Benchmarking

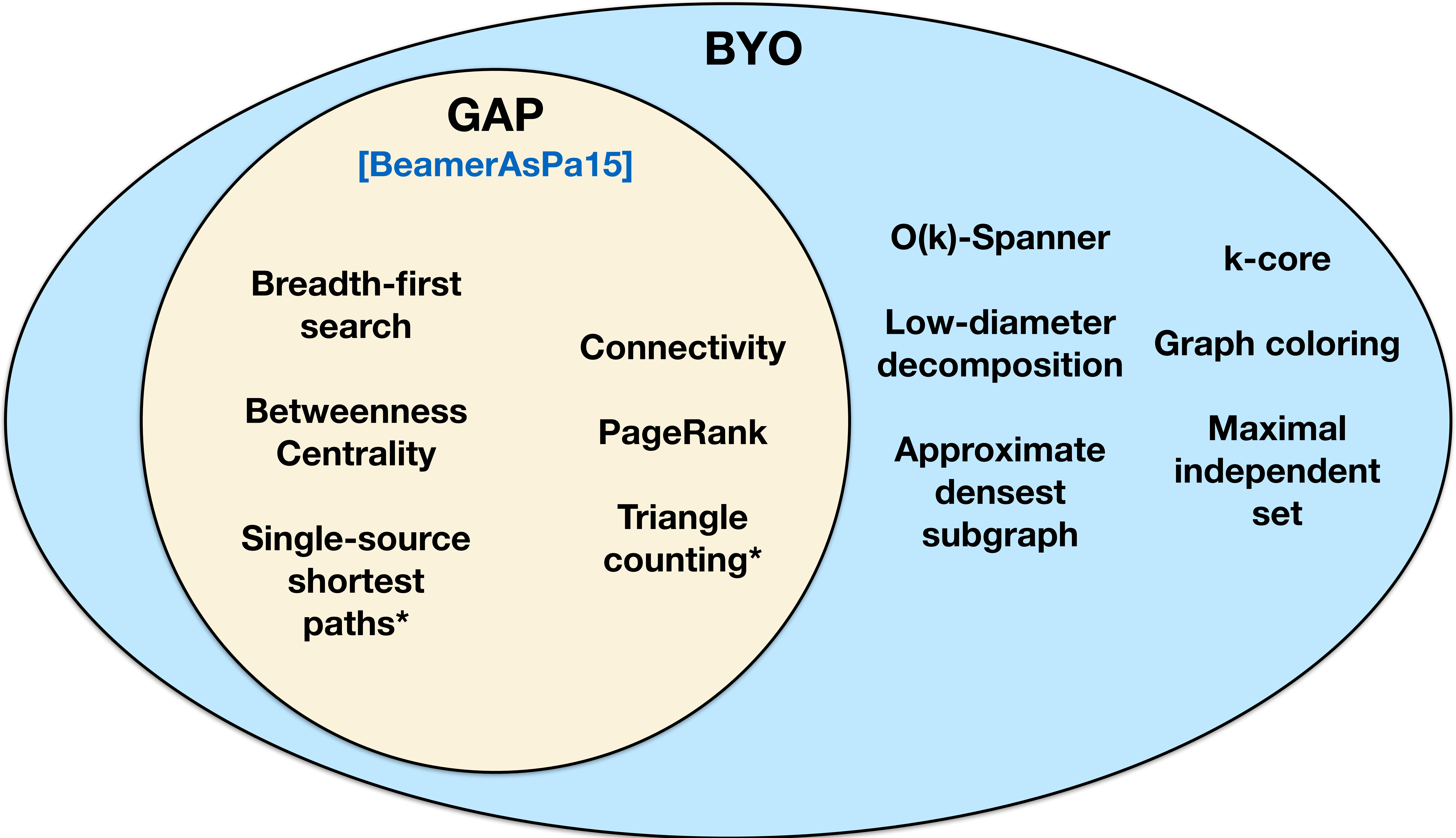


Table of Winners

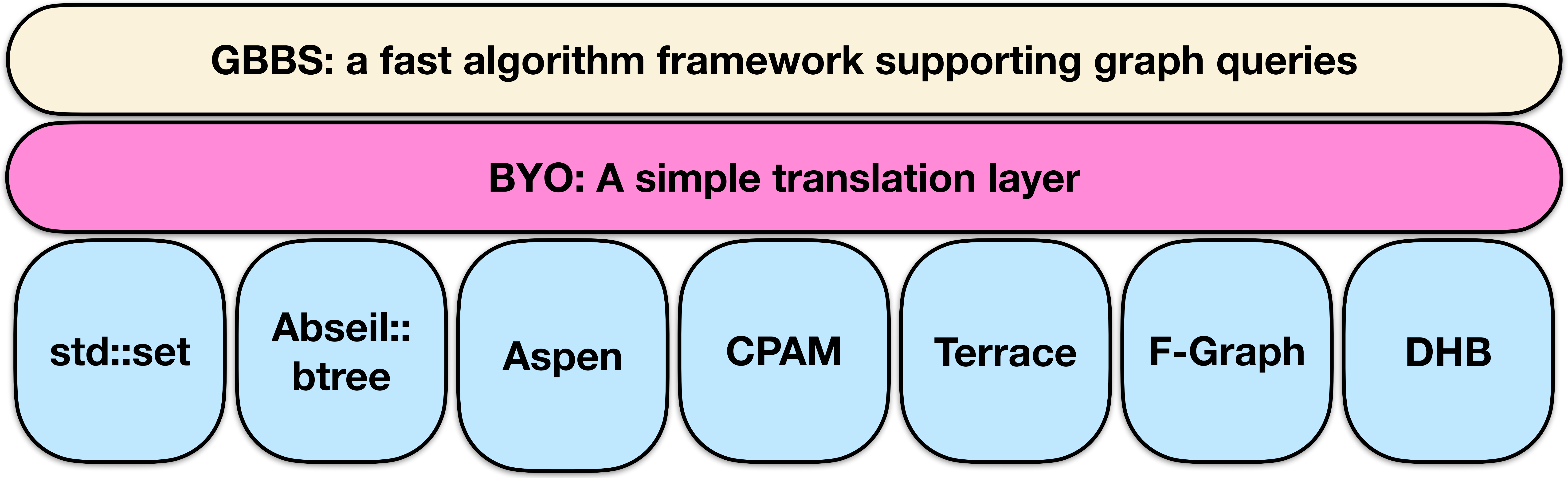
BYO enables users to ask the question "**what container is well-suited to which applications on which graphs**" without confounding factors from the framework/implementation details (e.g., parallelization framework, compiler, language, etc).

	RD	*LJ*	*CO*	*RM*	*ER*	*PR*	*TW*	*PA*	*FS*	*KR*
BFS	CPMA	CPAM*	Aspen*	CPAM*	CPAM*	TinySet	CPAM*	Aspen*	CPAM	CPMA
BC	absl::FHS*	CPAM*	Aspen*	CPAM*	CPAM*	DHB	Aspen*	Aspen*	Aspen*	CPAM*
Spanner	PMA	CPAM*	CPAM*	Aspen*	CPAM*	Aspen*	DHB	Aspen*	DHB	DHB
LDD	PMA	CPAM*	CPMA	CPAM*	CPAM*	DHB	CPMA	CPAM*	CPMA	CPMA
CC	absl::FHS*	Aspen*	Aspen	Aspen	Aspen	Aspen	Aspen	DHB	DHB	DHB
ADS	SSTGraph	TinySet	SSTGraph	Terrace	CPAM	absl::btree	PMA	DHB	DHB	DHB
KCore	C-CPAM*	C-CPAM*	CPAM	SSTGraph	SSTGraph	TinySet	CPAM	CPAM*	CPAM*	Aspen*
Coloring	PMA	PMA	PMA	PMA	PMA	PMA	PMA	PMA	PMA	PMA(V)
MIS	PMA	CPAM*	TinySet	Terrace	CPAM	TinySet	Aspen*	CPAM*	Aspen*	Aspen*
PR	DHB	Aspen*	Aspen	Terrace	CPAM*	Aspen	Aspen*	TinySet	PMA (V)	DHB

BYO Conclusion

BYO enables **apples-to-apples comparisons** between dynamic-graph containers by decoupling the graph container from the algorithm implementations.

The interface is simple, enabling **comprehensive comparisons** of new containers on a diverse set of applications with **minimal programming effort**.

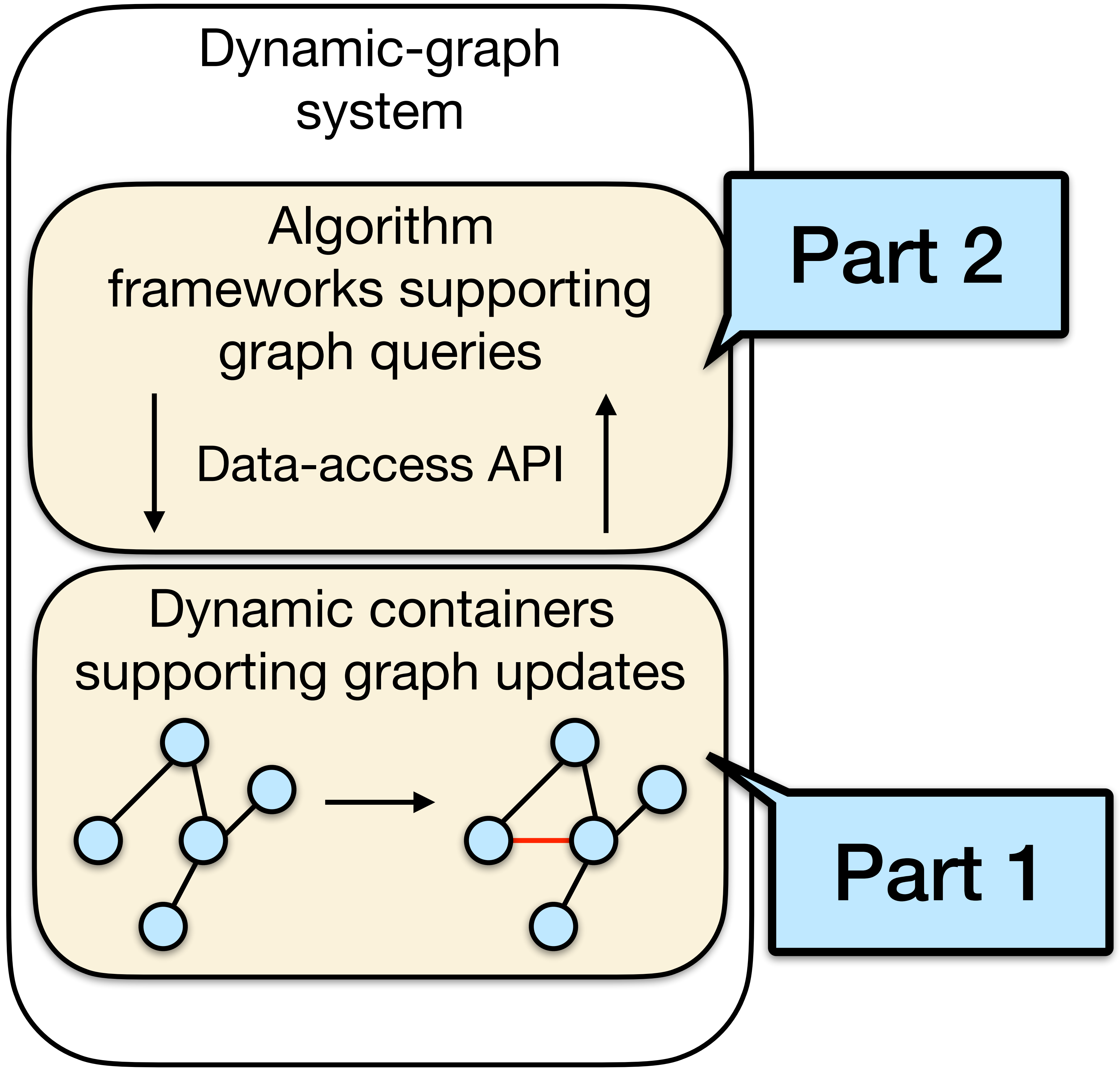


…and many others. Maybe your container is next? B(ring) Y(our) O(wn)

Code available at: https://github.com/wheatman/BYO

Conclusion

- Dynamic-graph **frameworks** and **containers** are both active areas of study.
- Despite the huge effort devoted to developing dynamic-graph systems (over 30+ papers in the past 10 years), at present, it is hard to tell **which system (i.e., which framework and container) is best for a given workload**.
- **Standardizing evaluations with frameworks** is an important step in answering this question.



Where to go from here…

- **Associating large vector embeddings with nodes/edges in the graph**
  - Common in graph-based vector databases, Graph neural networks, etc.
  - The data transfer bottleneck changes from graph structure to associated data
- **Designing a framework for evaluating graph systems that concurrently run graph algorithms and updates**
  - Guaranteeing serializability is challenging in the presence of long running graph algorithms and updates
  - Graph databases propose transaction-based solutions (MVCC)
  - Several papers on developing custom solutions for specific graph algorithms

BACKUP

Graph sizes

Graph	*Vertices*	*Edges*	*Avg. Degree*
Road (RD)	23,947,347	57,708,624	2
LiveJournal (LJ)	4,847,571	85,702,474	18
Com-Orkut (CO)	3,072,627	234,370,166	76
rMAT (RM)	8,388,608	563,816,288	67
Erdős-Rényi (ER)	10,000,000	1,000,009,380	100
Protein (PR)	8,745,543	1,309,240,502	150
Twitter (TW)	61,578,415	2,405,026,092	39
papers100M (PA)	111,059,956	3,228,124,712	29
Friendster (FS)	124,836,180	3,612,134,270	29
Kron (KR)	134,217,728	4,223,264,644	31

Table 3: Sizes of (symmetrized) graphs used (ordered by size).

Container	*Slowdown over CSR*			*Bytes per edge*		
	Average	*95%*	*Max*	*Min*	*Average*	*Max*
NeighborSet API (Vector of...)						
`absl::btree_set`	1.26	1.9	2.3			
`absl::btree_set` (inline)	1.22	2	2.6			
`absl::flat_hash_set`	1.40	2.3	3.4			
`absl::flat_hash_set` (inline)	1.29	2.1	2.6			
`std::set`	2.59	5.0	5.8			
`std::set` (inline)	2.37	4.9	5.6			
`std::unordered_set`	2.01	3.7	6.0			
`std::unordered_set` (inline)	1.90	3.5	5.9			
Aspen	1.22	2	2.5	5.7	12.0	53.4
Aspen (inline)	1.14	1.7	2.0	5.8	7.4	14.9
Compressed Aspen	1.44	2.1	2.6	3.4	5.0	12.1
Compressed Aspen (inline)	1.34	1.9	2.6	3.4	5.5	14.9
CPAM	1.16	1.4	1.5	4.1	4.9	9.0
CPAM (inline)	1.11	1.5	1.6	4.1	6.6	21.6
Compressed CPAM	1.37	1.7	1.9	3.4	4.5	8.9
Compressed CPAM (inline)	1.30	1.8	2.1	3.5	6.2	21.6
PMA	1.25	1.9	3.2	8.1	13.9	46.5
Compressed PMA	1.35	1.9	3.3	4.9	11.2	46.5
Tinyset	1.27	1.9	5.1	5.5	8.6	26.5
Vector	1.07	1.4	1.9	4.1	5.0	10.2
GraphContainer API						
CSR	1.00	1.0	1.0	4.1	5.1	10.6
Compressed CSR	1.23	1.5	1.6	2.3	3.8	10.6
DHB	1.15	1.7	2.4			
PMA	1.15	1.4	1.6	10.0	12.3	24.2
Compressed PMA	1.31	2.0	2.2	3.1	5.6	17.7
SSTGraph	1.25	1.5	2.4	4.0	6.4	19.9
Terrace	1.20	2.0	3.3	9.3	17.7	47.8

Table 5: Data structure algorithm performance and space usage. All data structures are uncompressed unless otherwise specified. Each container's time is normalized to CSR's time averaged over all 100 settings of 10 algorithms × 10 graphs. A number closer to 1 means better performance (higher is worse). The 95% and max columns show the 95th percentile and maximum slowdown over CSR across

API configuration	*Slowdown over full API*		
	Average	*95%*	*Max*
Min (just `map_neighbors` and `num_vertices`)	10.69	231	1379
Min + `degree`	1.43	4.1	22.8
Min efficient (Min + `degree` + `num_edges`)	1.16	2.5	3.1
Full minus `num_edges`	1.31	2.78	22.9
Full minus `degree`	2.18	7.4	14.5
No early exit (Full minus both map early exit)	1.12	2.5	3.1
No parallel map (Full minus both parallel map)	1.01	1.3	1.9
Full minus `parallel_map_neighbors_early_exit`	0.98	1.03	1.1
Full (All required and optional functionality)	1.00	1.00	1.00

Table 4: The average performance of different GraphContainer API configurations with CSR as the underlying container. "Min" refers BYO with just the required functionality and "Full" refers to BYO with both required and optional functions described in Section 4.2. The remaining configurations are described with either what they add to min, or what they remove from full. The 95% and max columns show the 95th percentile and maximum slowdown over the full API across all algorithms and all graphs. Configurations that achieve within $1.25\times$ slowdown over the full API are shaded.