Reducing the Storage Overhead of Main-Memory OLTP Databases with **Hybrid Indexes**

Huanchen Zhang

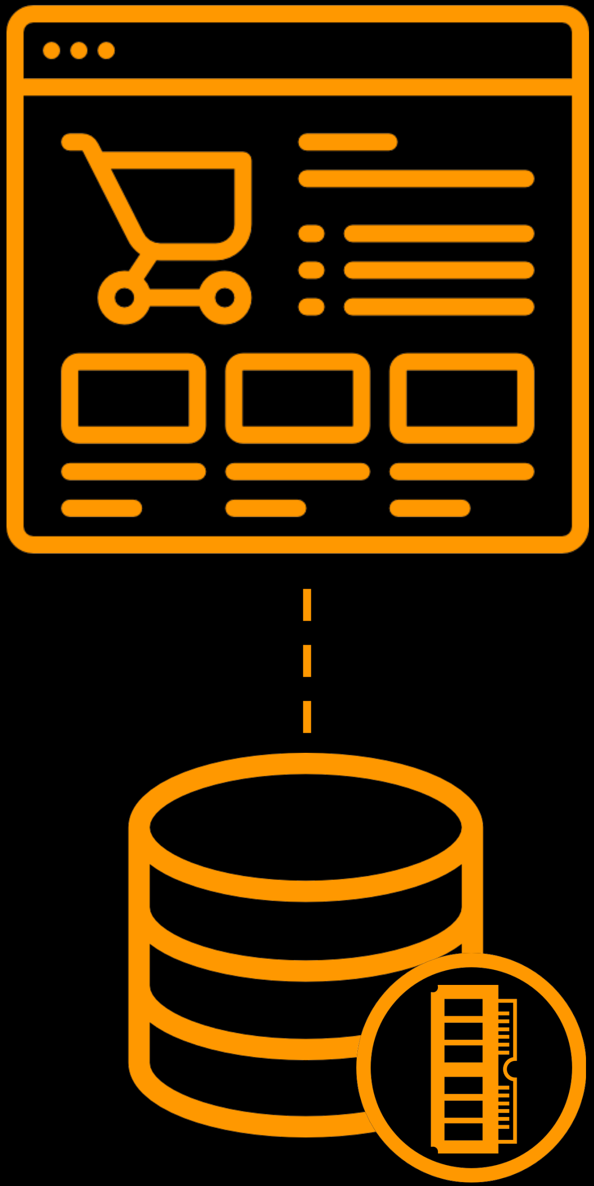
David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, Rui Shen

PARALLEL DATA LABORATORY

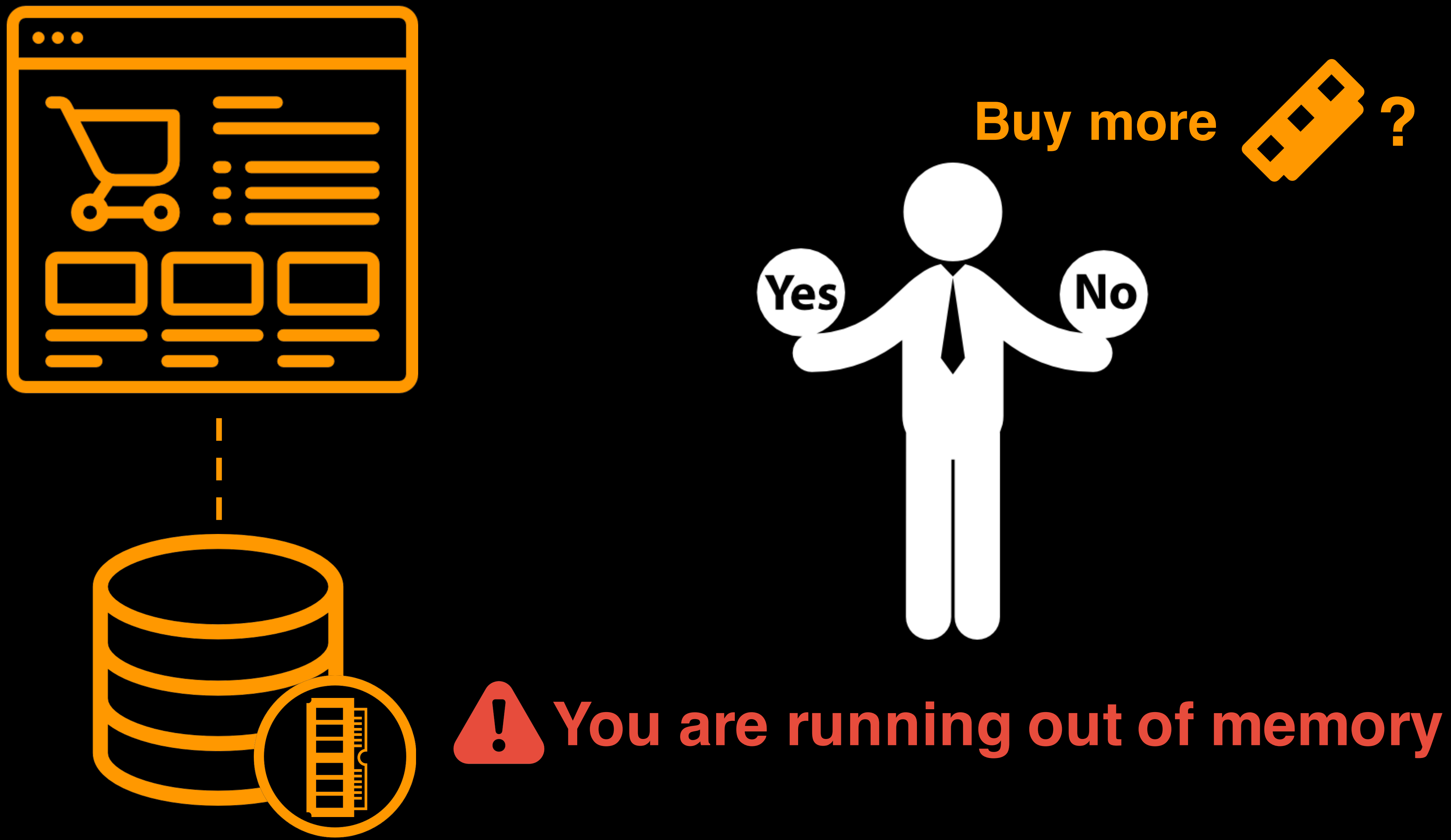
Carnegie Mellon University

Part I
Initial Exploration
of Hybrid Indexes
[SIGMOD'16]

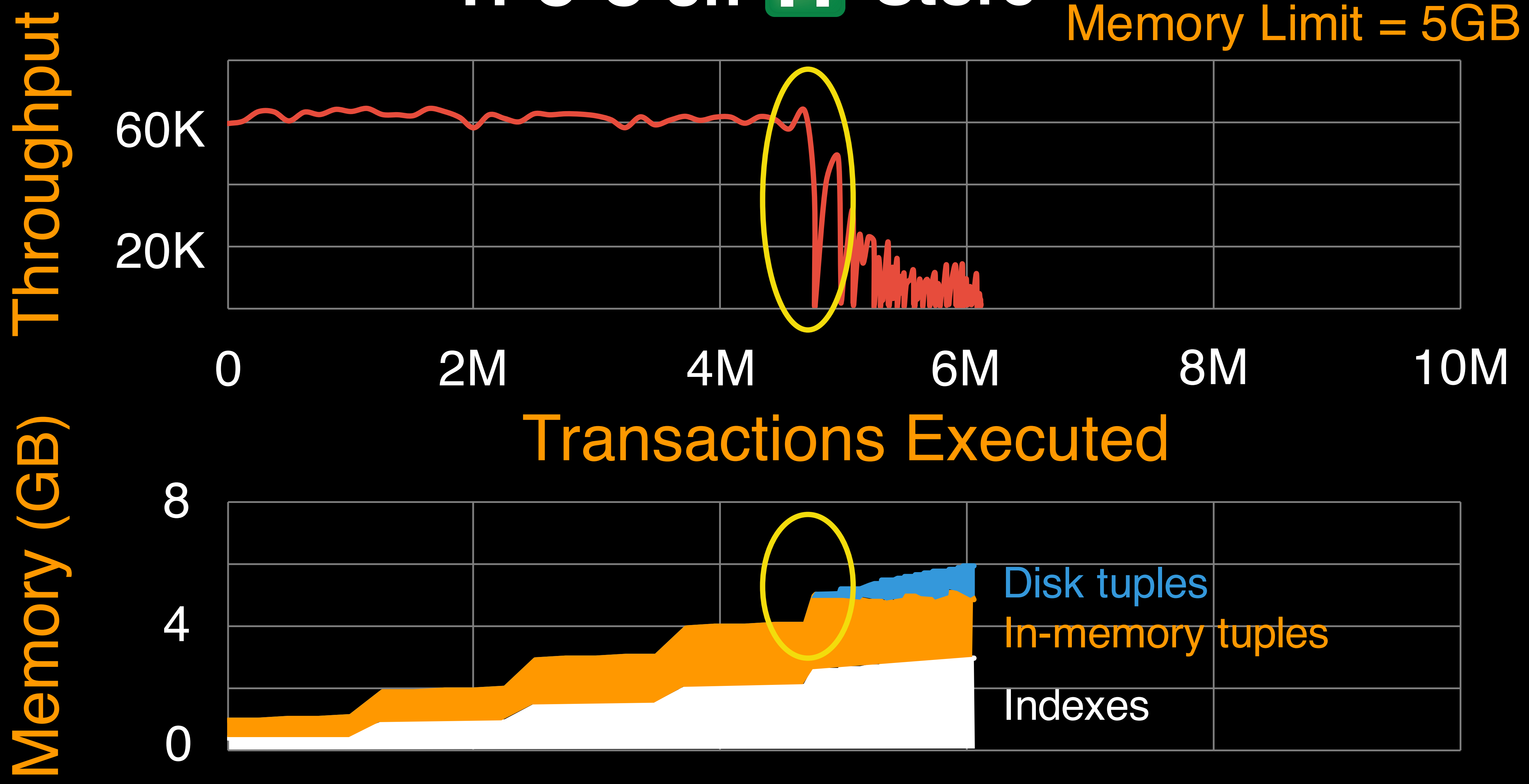

You are running out of memory



You are running out of memory


Buy more ?
Yes
No
You are running out of memory

TPC-C on H-Store




I GOT STUCK
SO I WENT TO SLEEP

The better way:
Use memory more efficiently

Indexes are LARGE

Benchmark	% space for index		Hybrid Index
TPC-C	58%	→	34%
Voter	55%	→	41%
Articles	34%	→	18%

Our Contributions [SIGMOD'16]

1 The hybrid index architecture

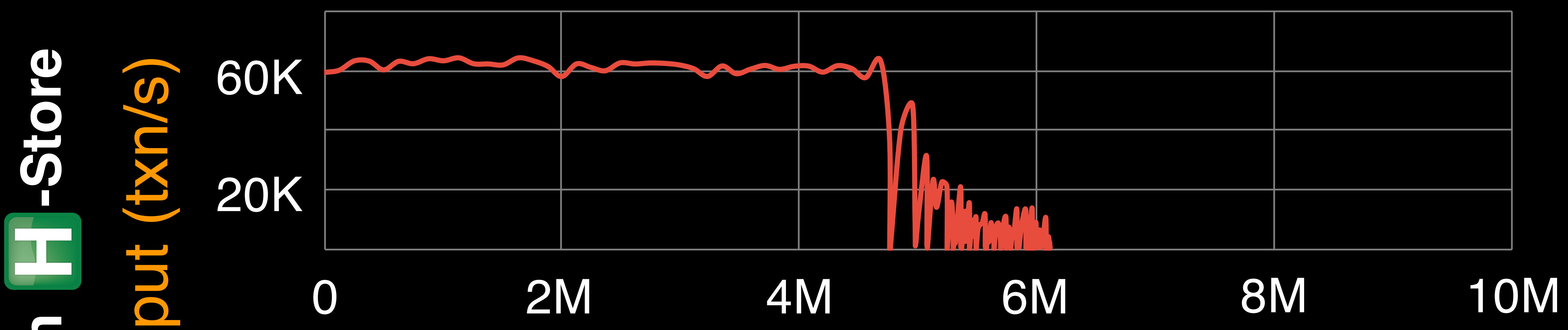
2 The Dual-Stage Transformation

3 Applied to 4 index structures

- B+tree
- Masstree
- Skip List
- Adaptive Radix Tree (ART)

Performance	**Space**
$\approx$	**30 – 70%** ↓

Did we solve this problem?



Stay tuned

How do hybrid indexes achieve memory savings ?

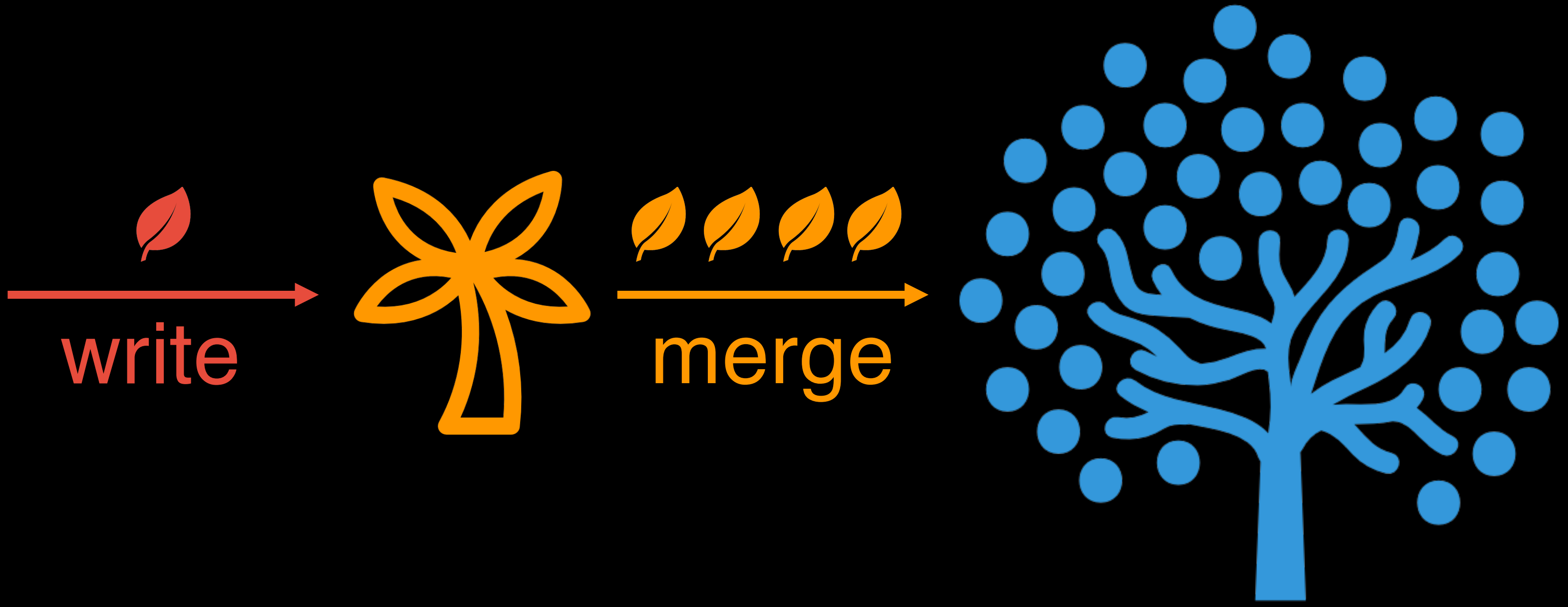
Static

Hybrid Index: a dual-stage architecture

dynamic stage

static stage

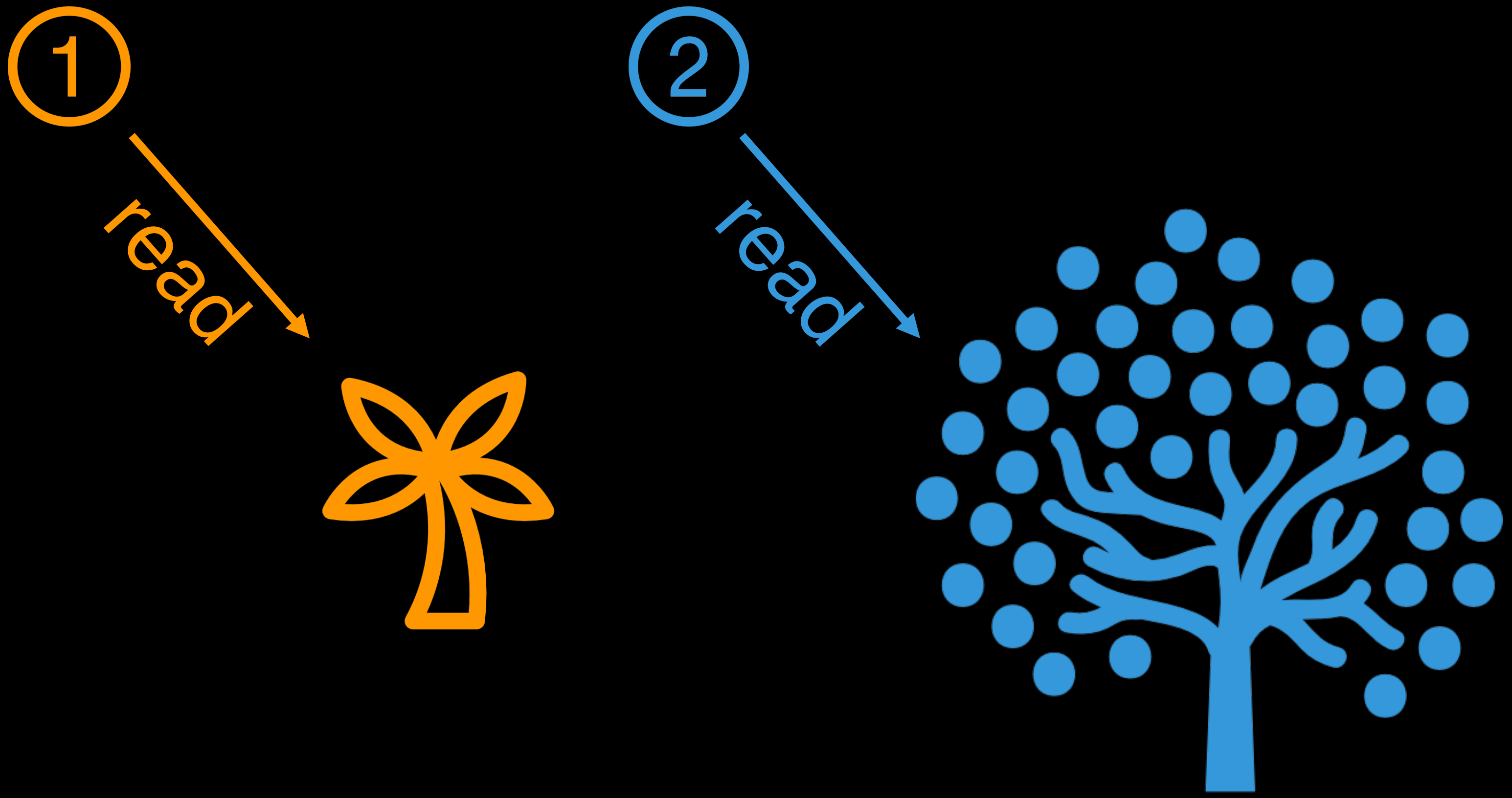
Inserts are batched in the dynamic stage



dynamic stage

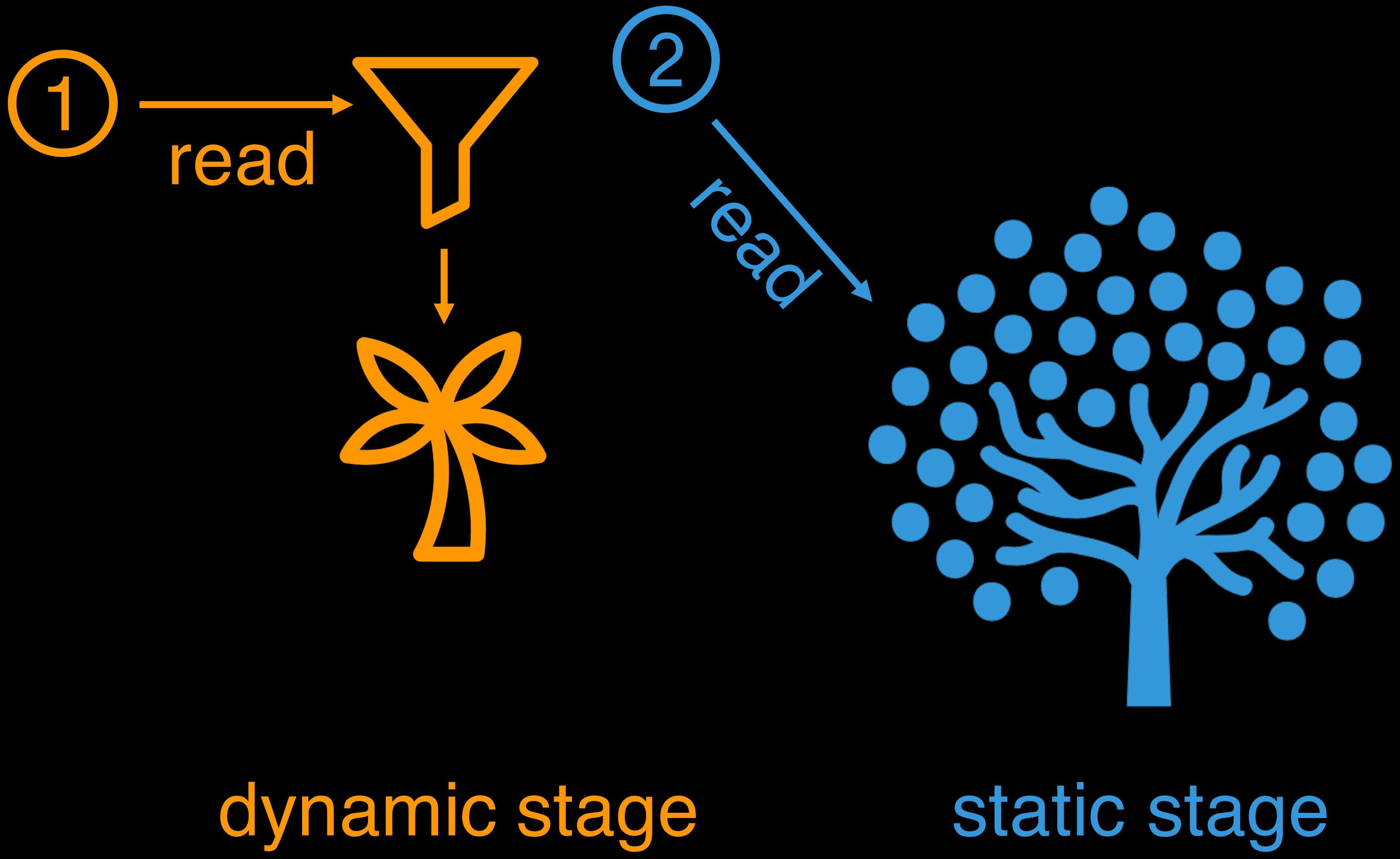
static stage

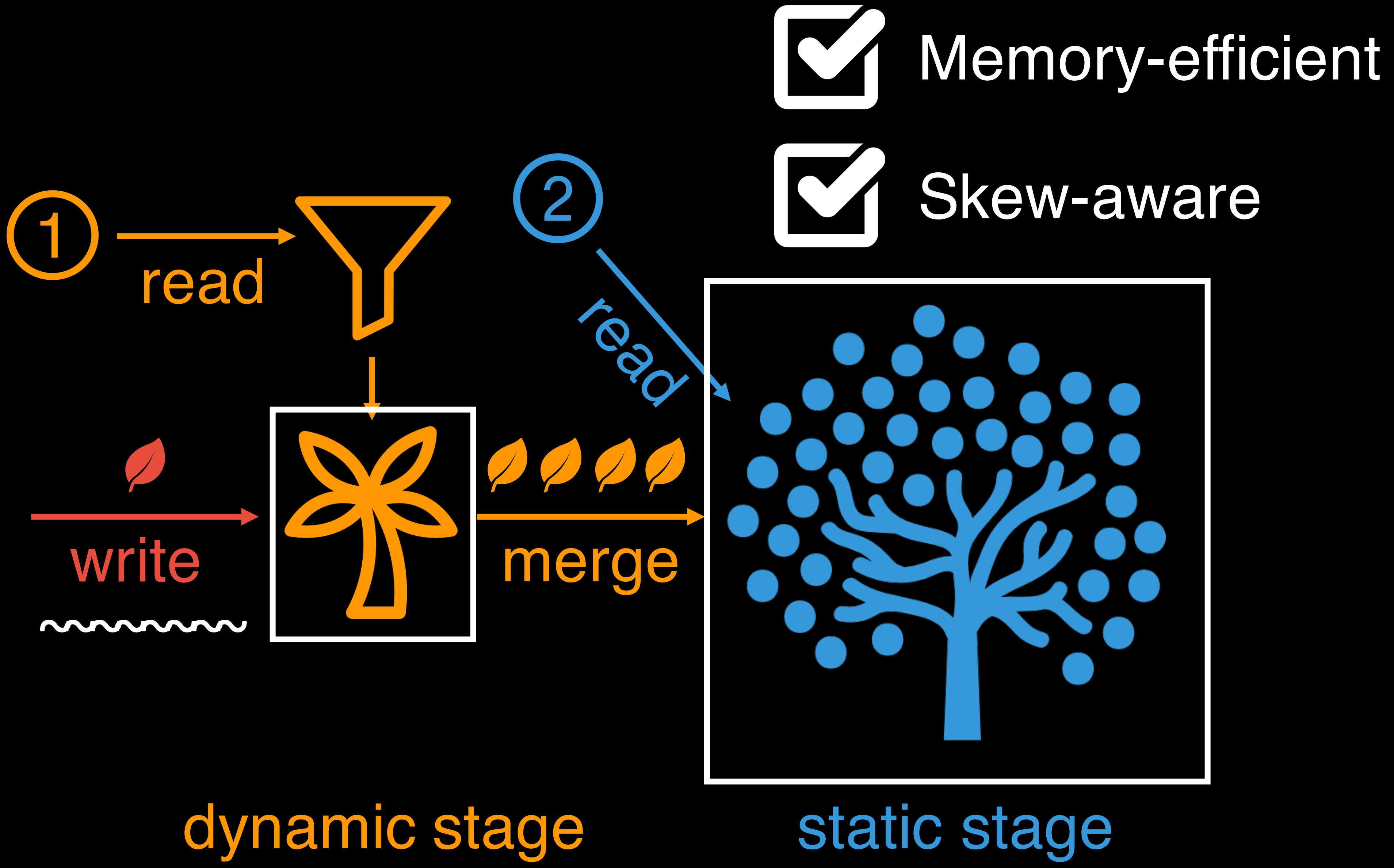
Reads search the stages in order



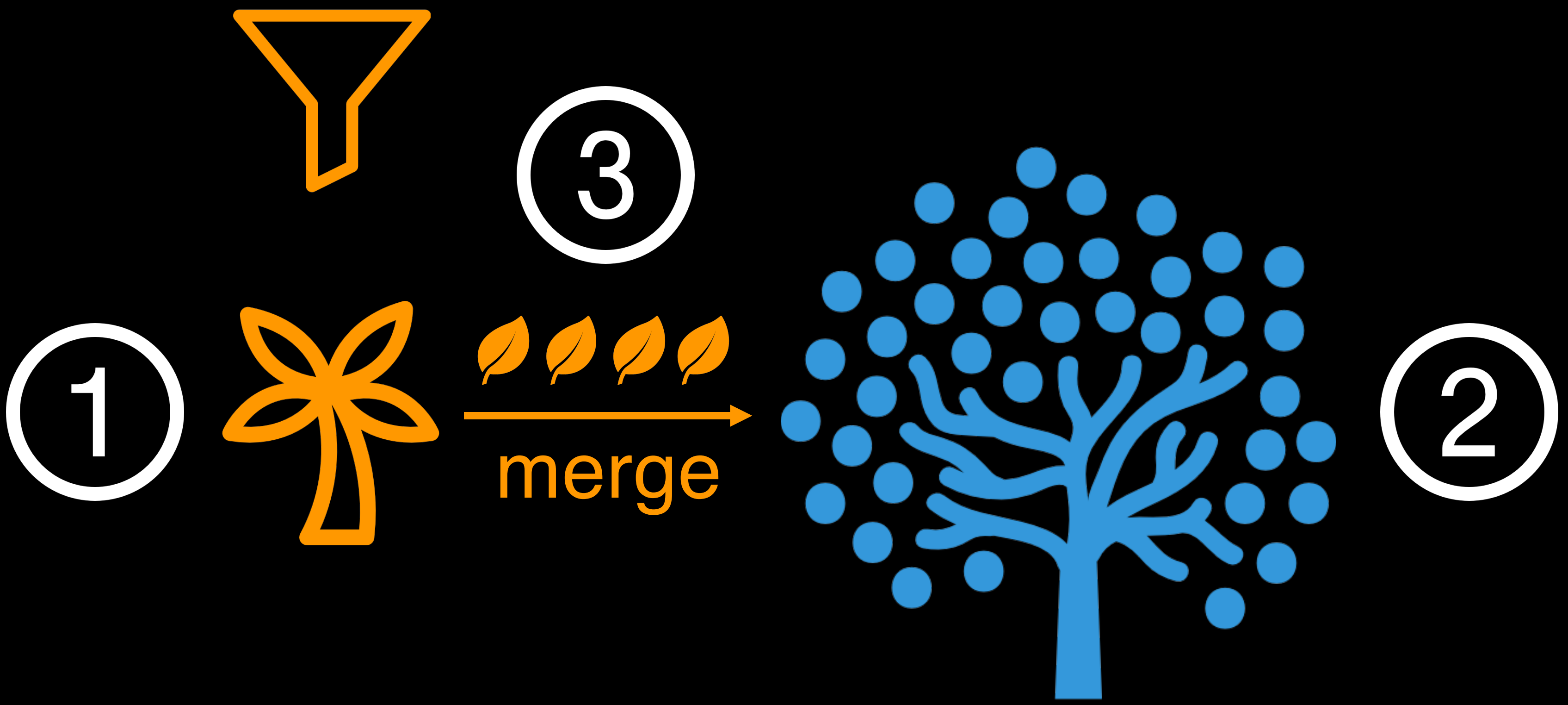
dynamic stage static stage

A Bloom filter improves read performance




Memory-efficient
Skew-aware
1
read
2
read
write
merge
dynamic stage
static stage

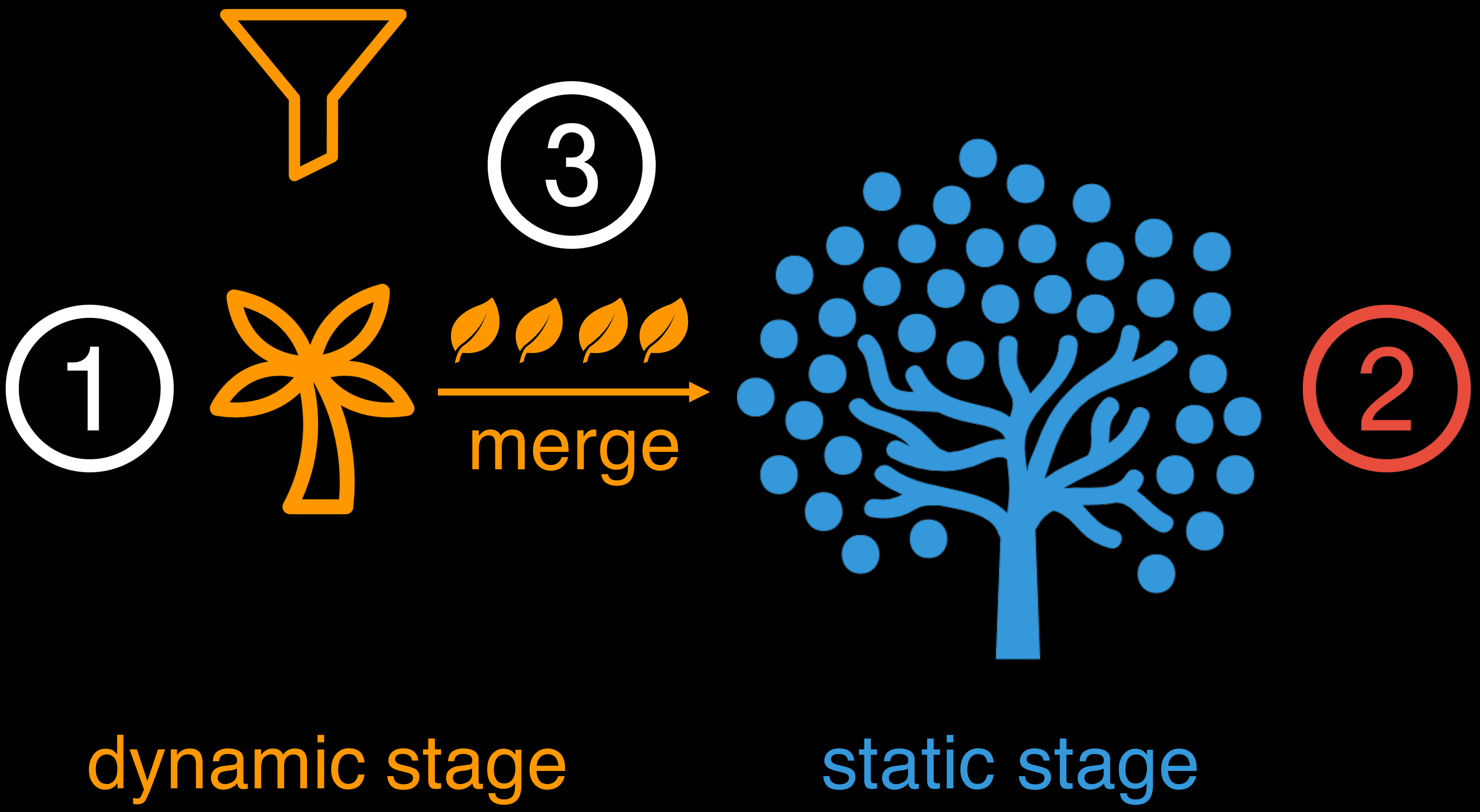
The Dual-Stage Transformation



dynamic stage

static stage

The Dual-Stage Transformation



The Dynamic-to-Static Rules

Compaction

Reduction

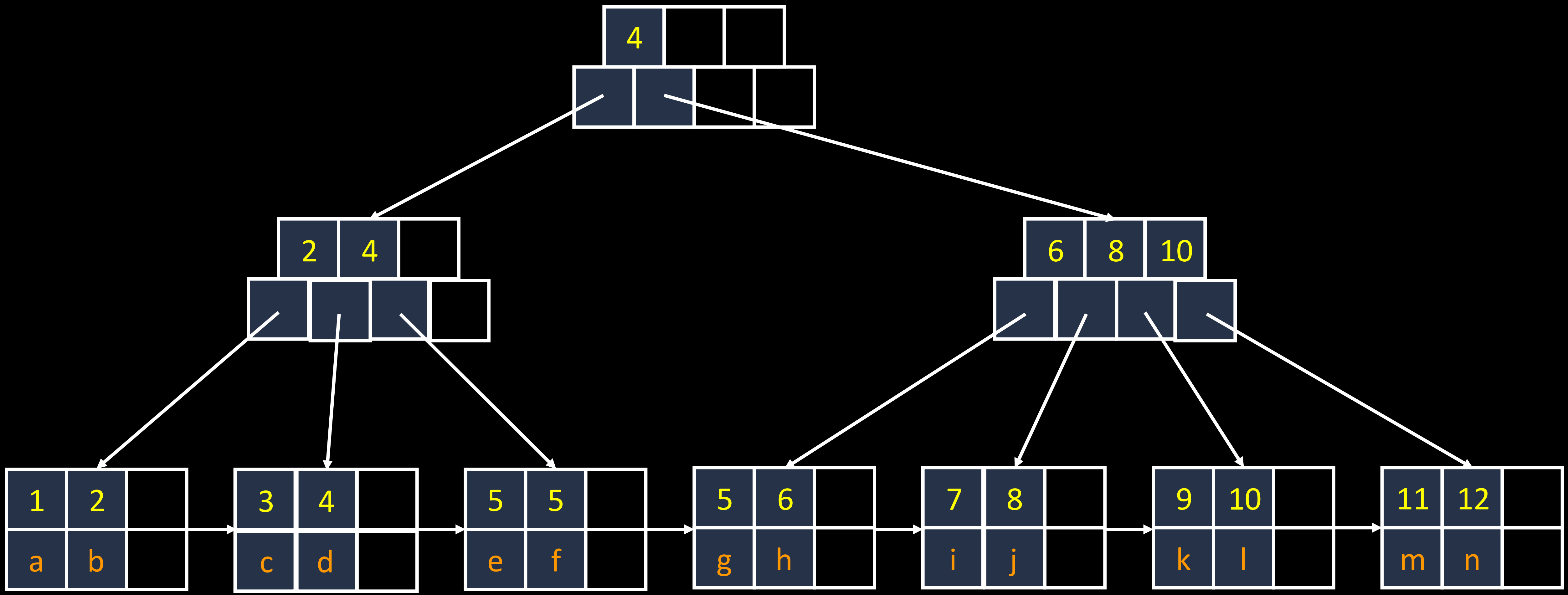
Compression

The Dynamic-to-Static Rules

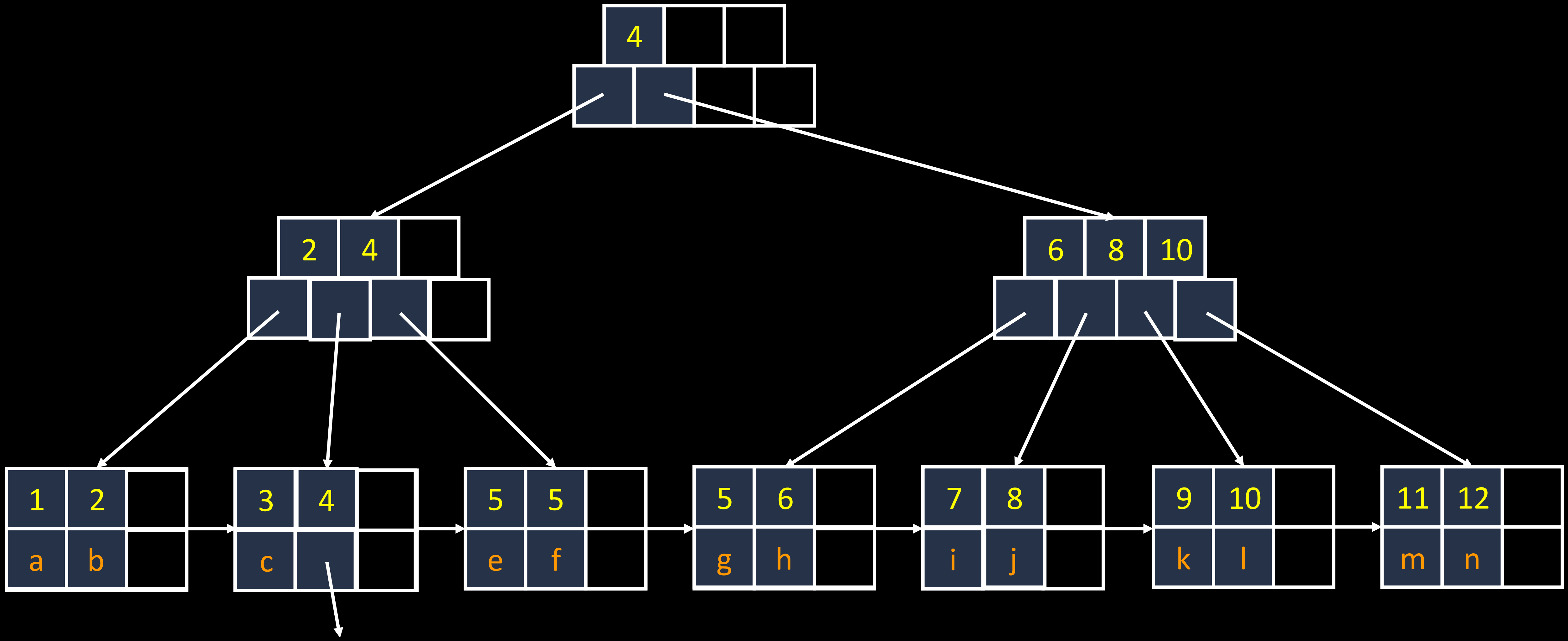
Compaction

Reduction

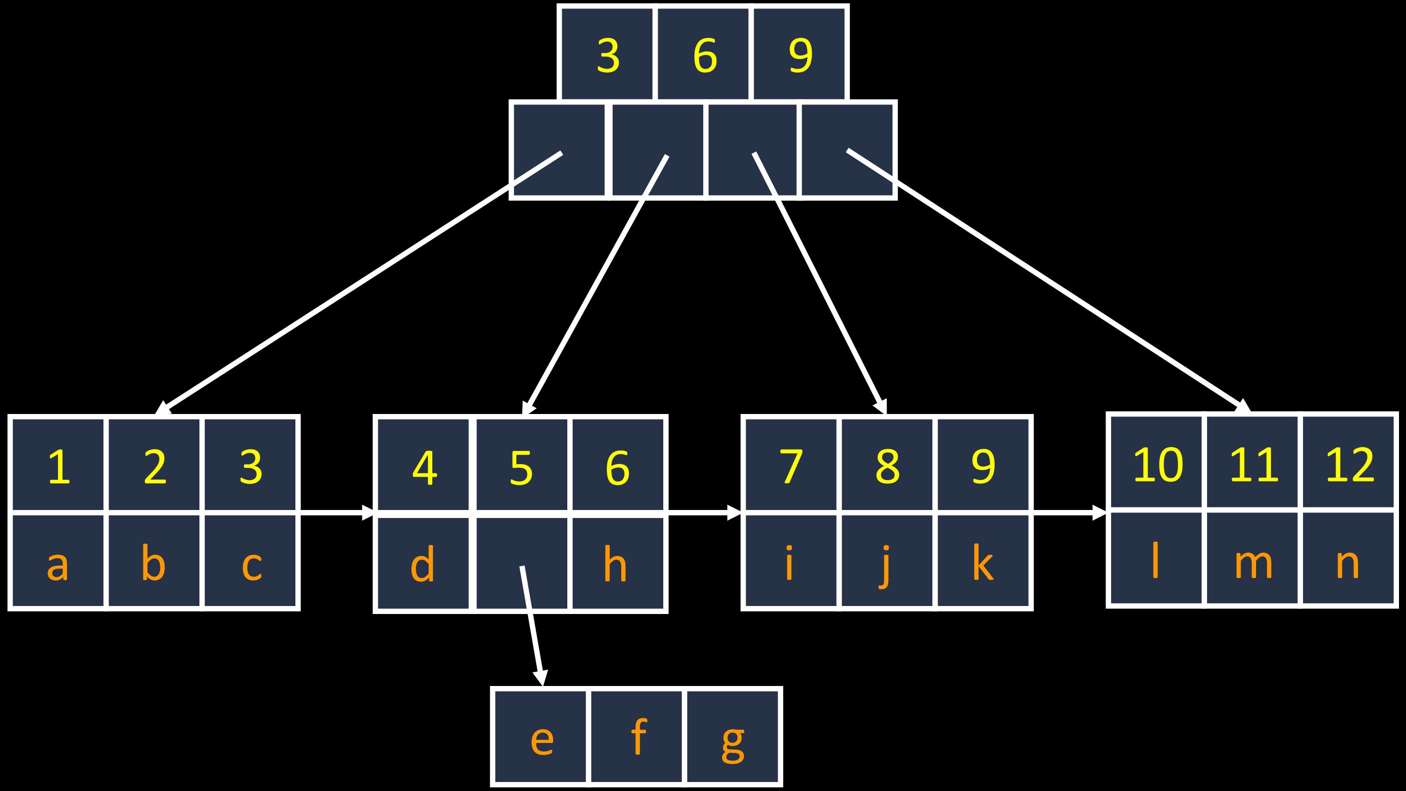
Compression


4
2 4
6 8 10
1 2 a b
3 4 c d
5 5 e f
5 6 g h
7 8 i j
9 10 k l
11 12 m n

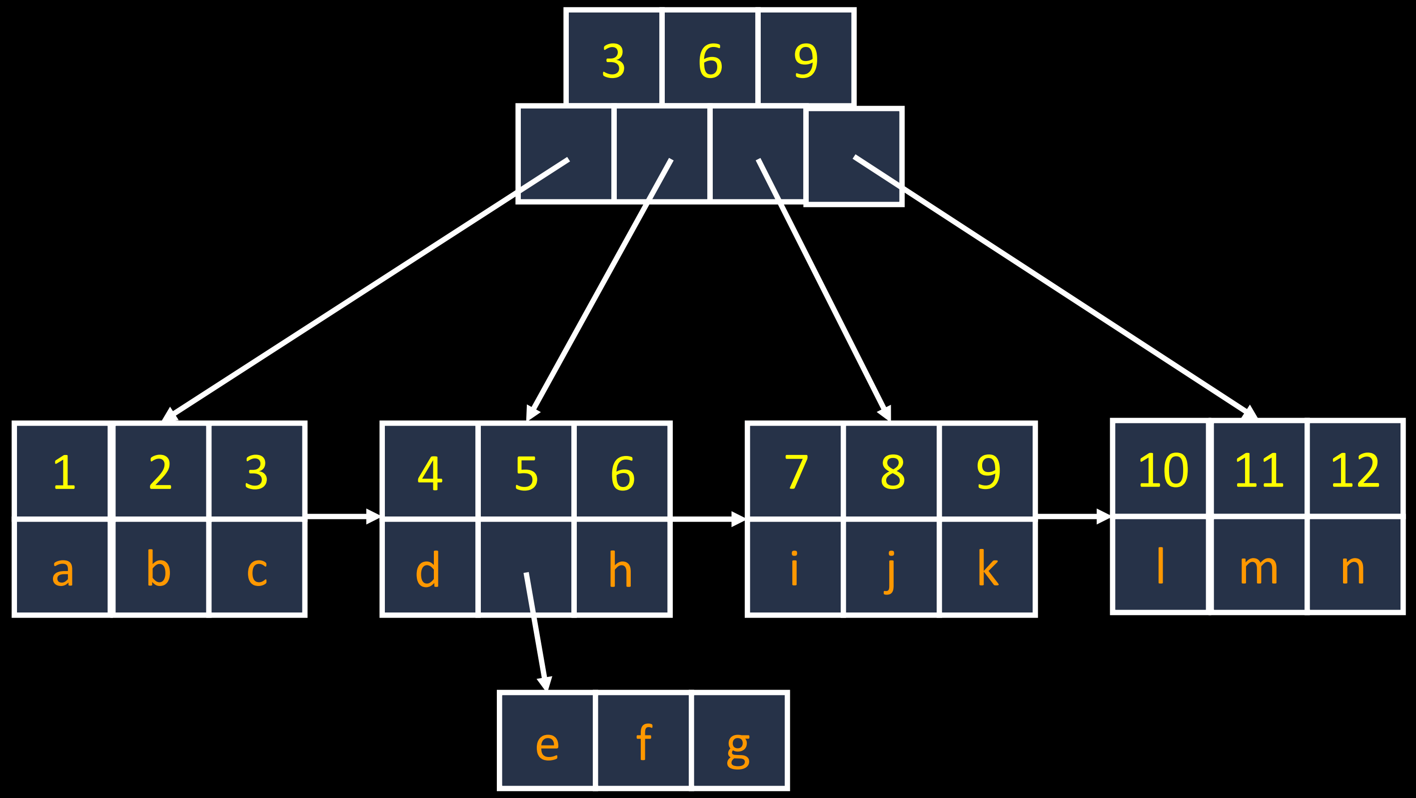
Compaction: minimize # of memory blocks



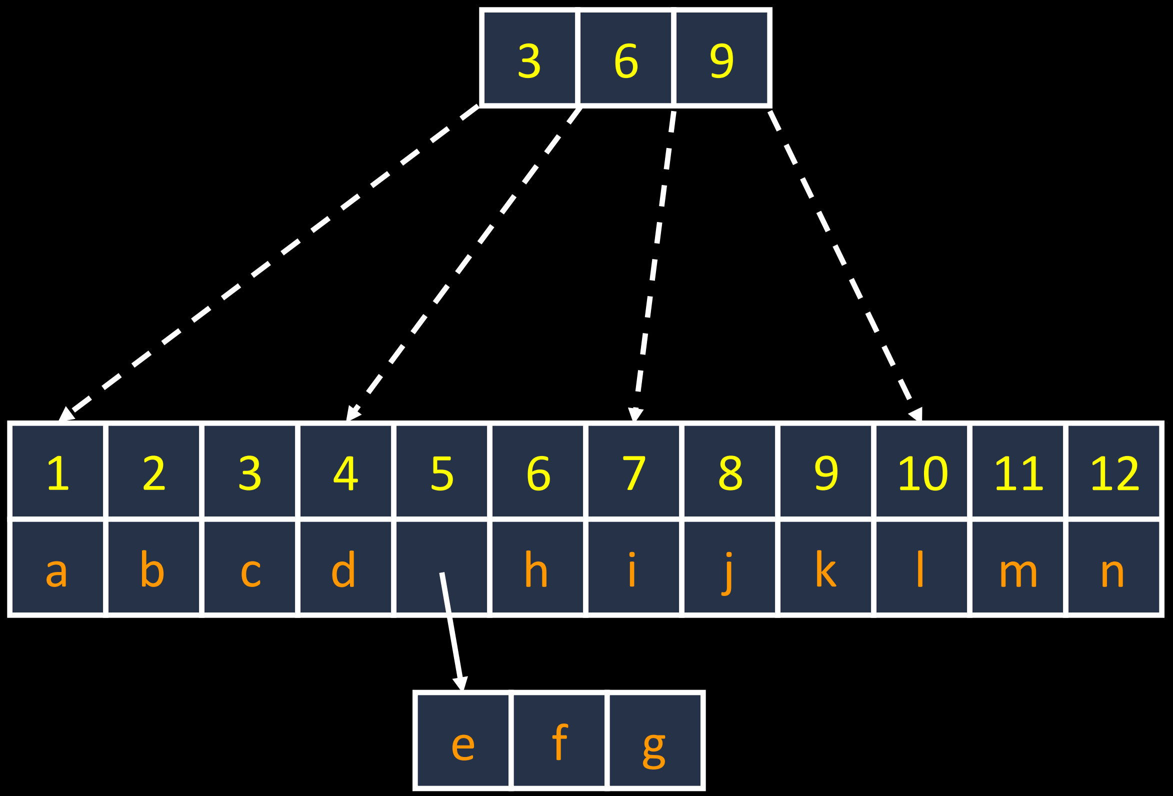
Compaction: minimize # of memory blocks



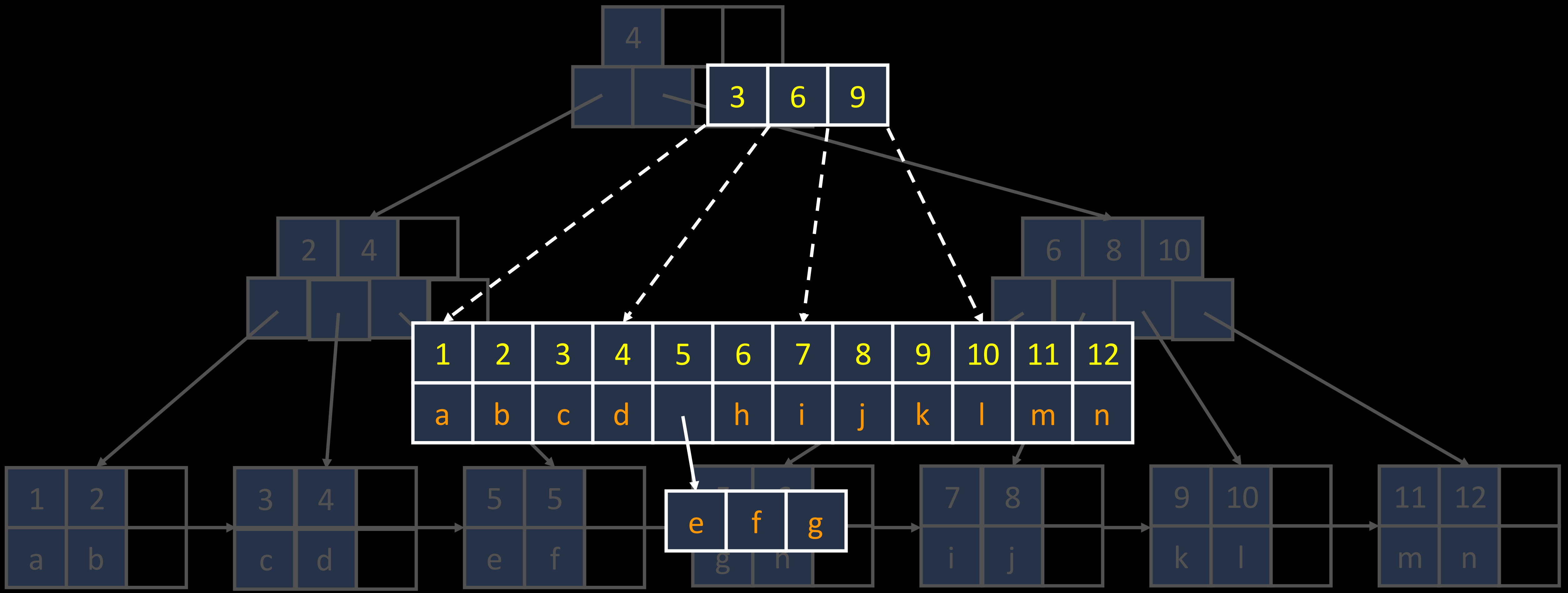
Reduction: minimize structural overhead



Reduction: minimize structural overhead

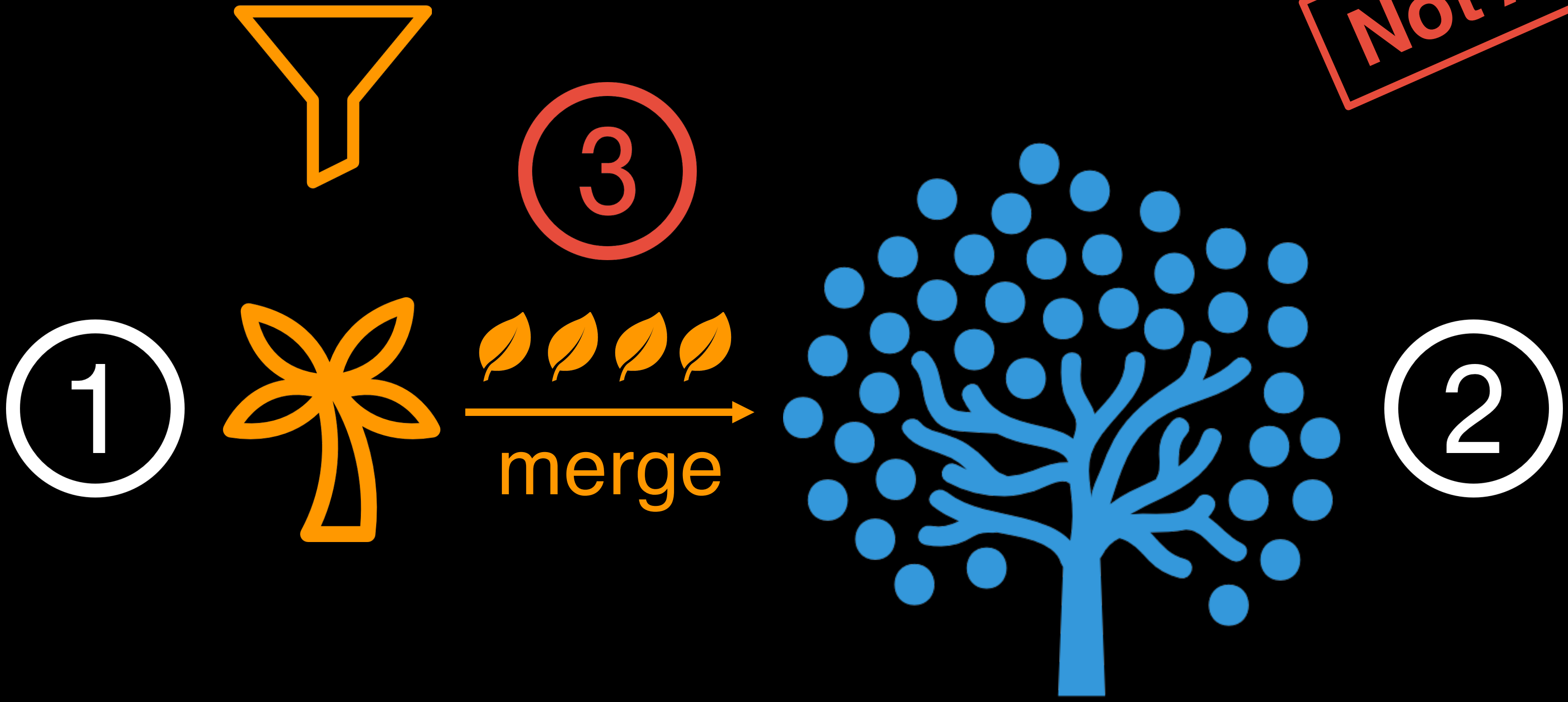


Reduction: minimize structural overhead



The merge routine is a blocking process

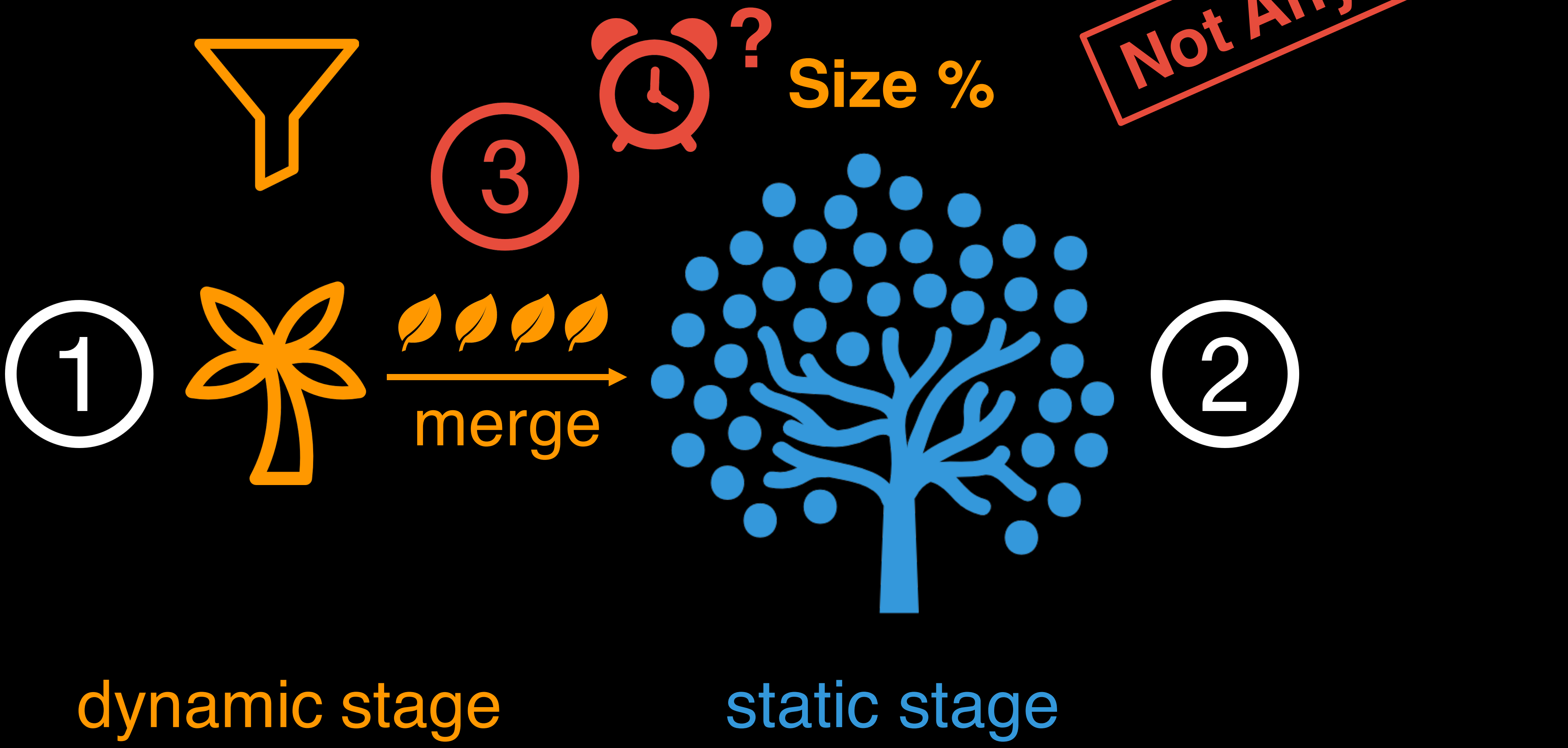
Not Any More



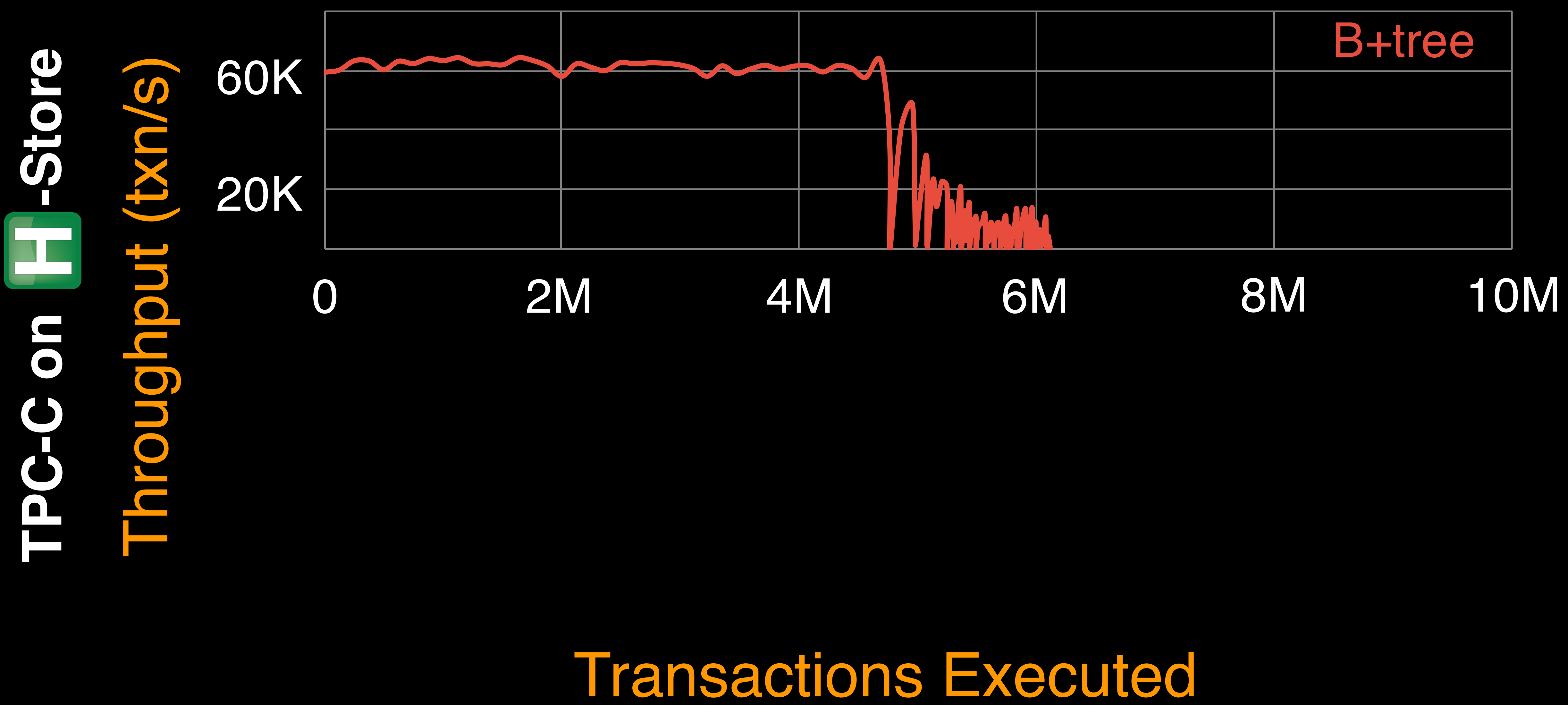
dynamic stage

static stage

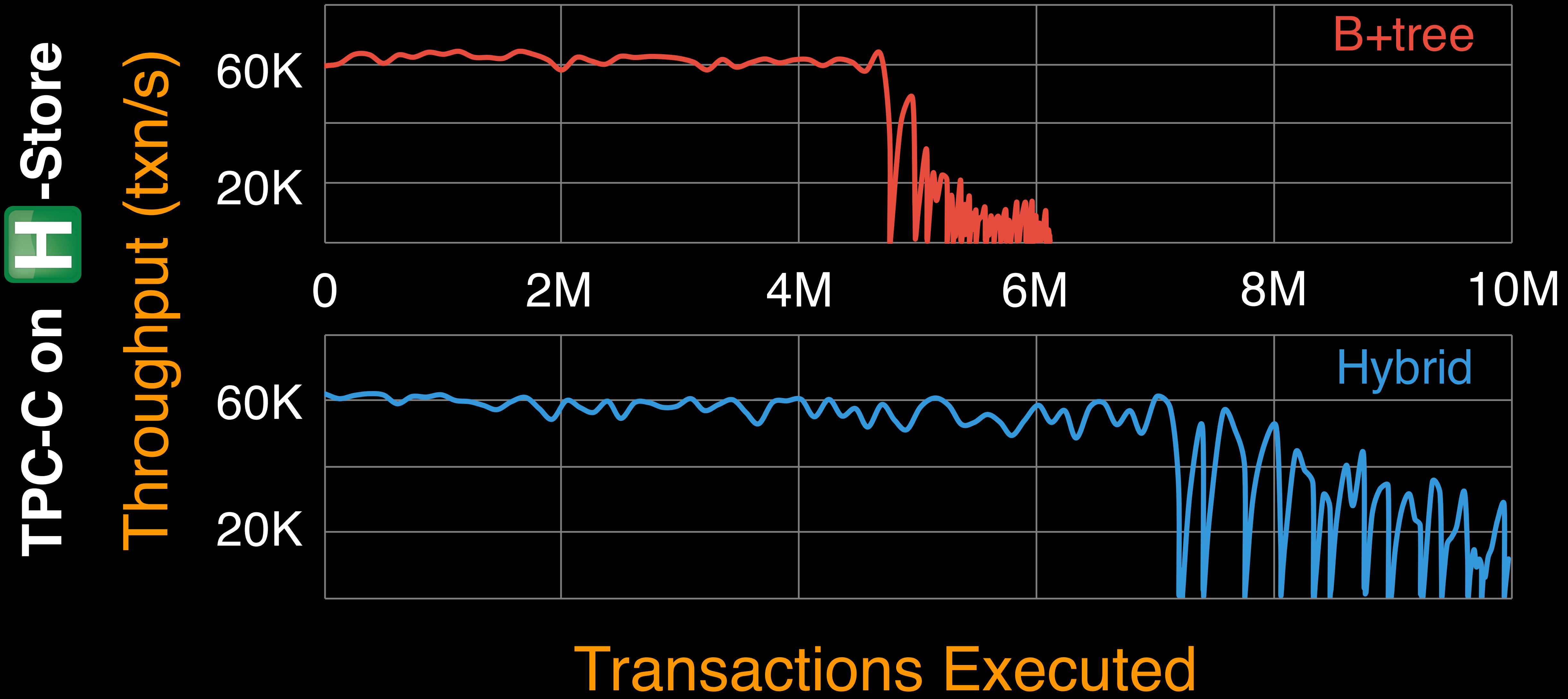
The merge routine is a blocking process



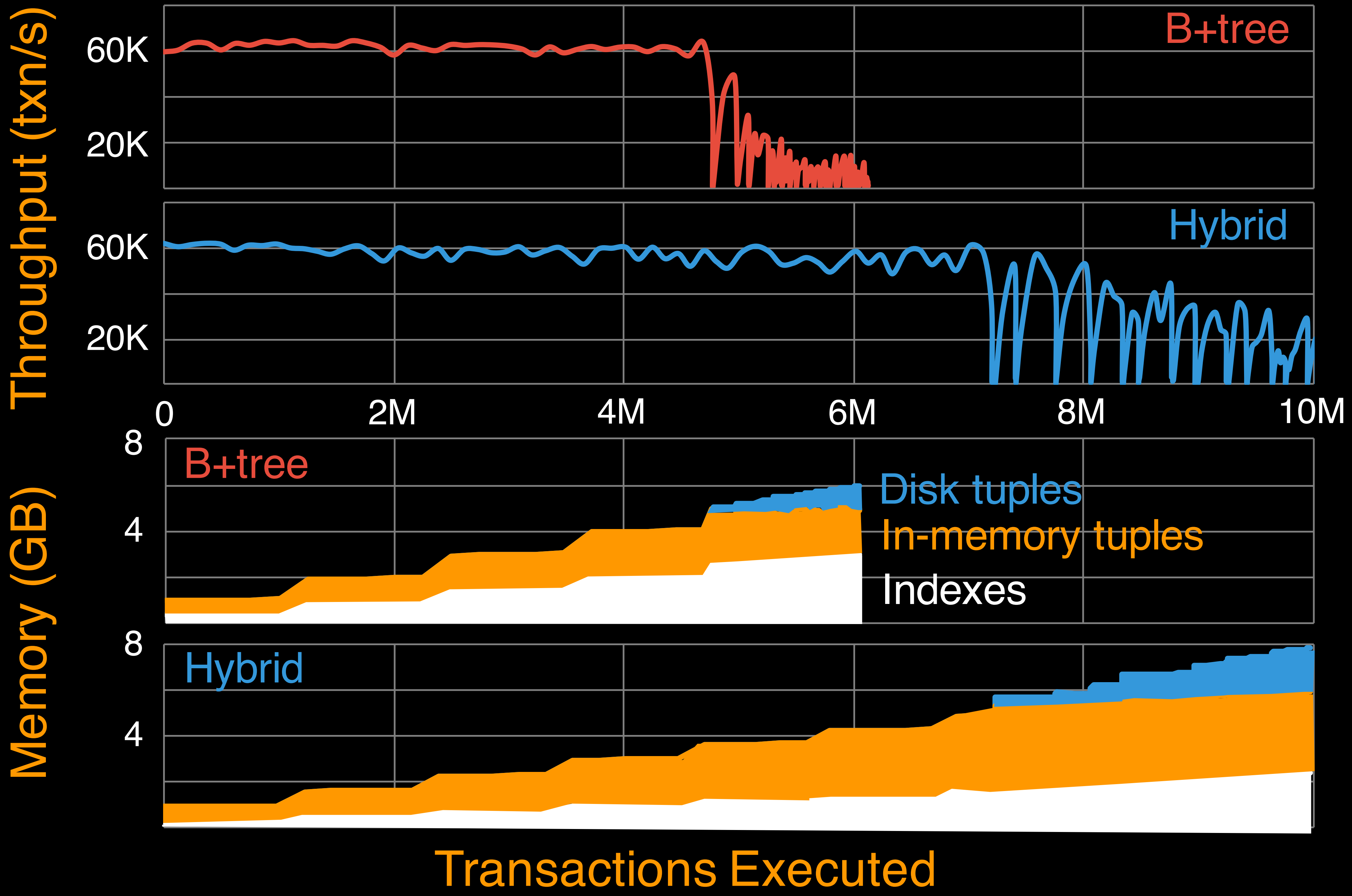
Did we solve this problem?



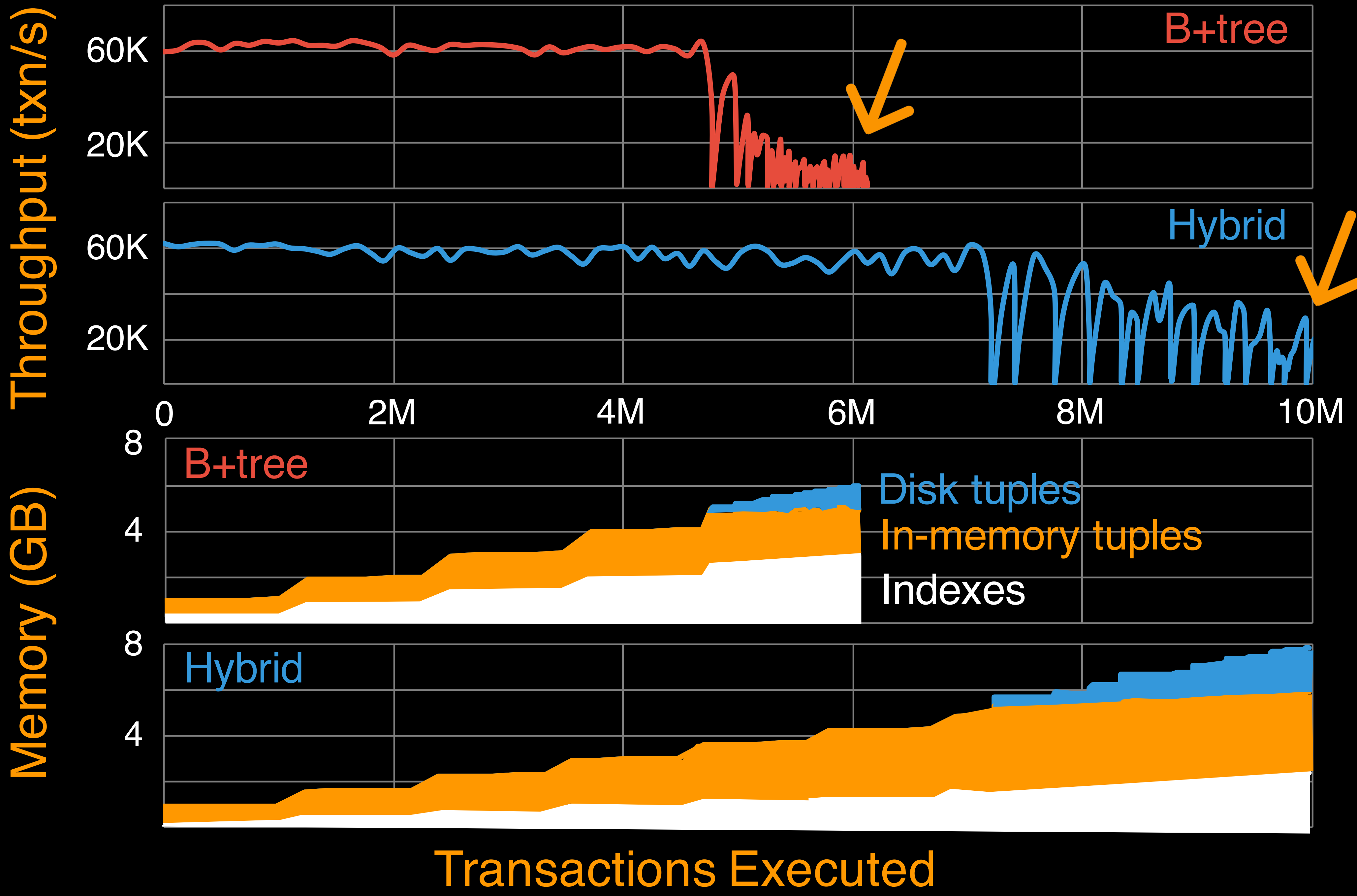
Yes, we improved the DBMS's capacity!

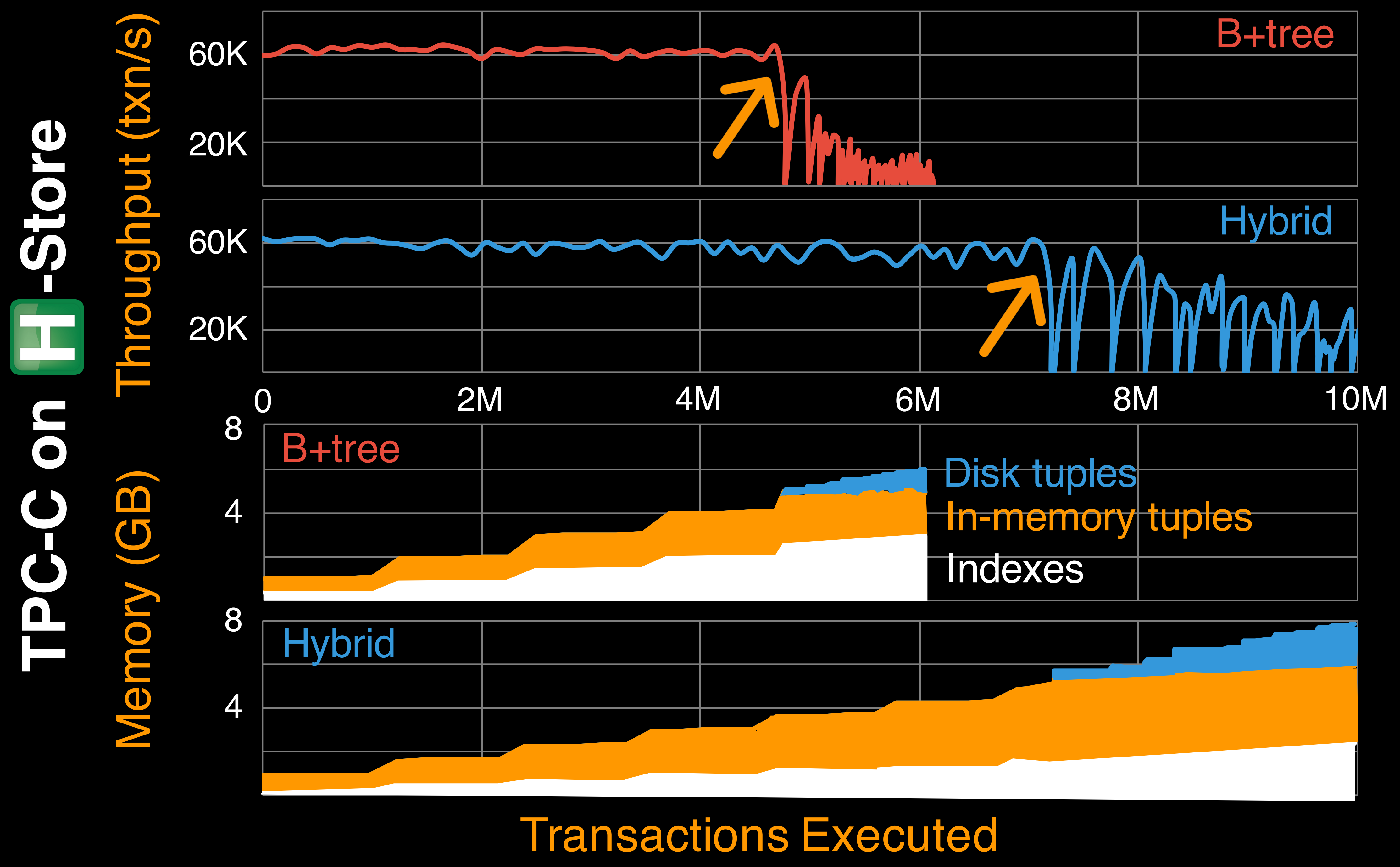


TPC-C on H-Store

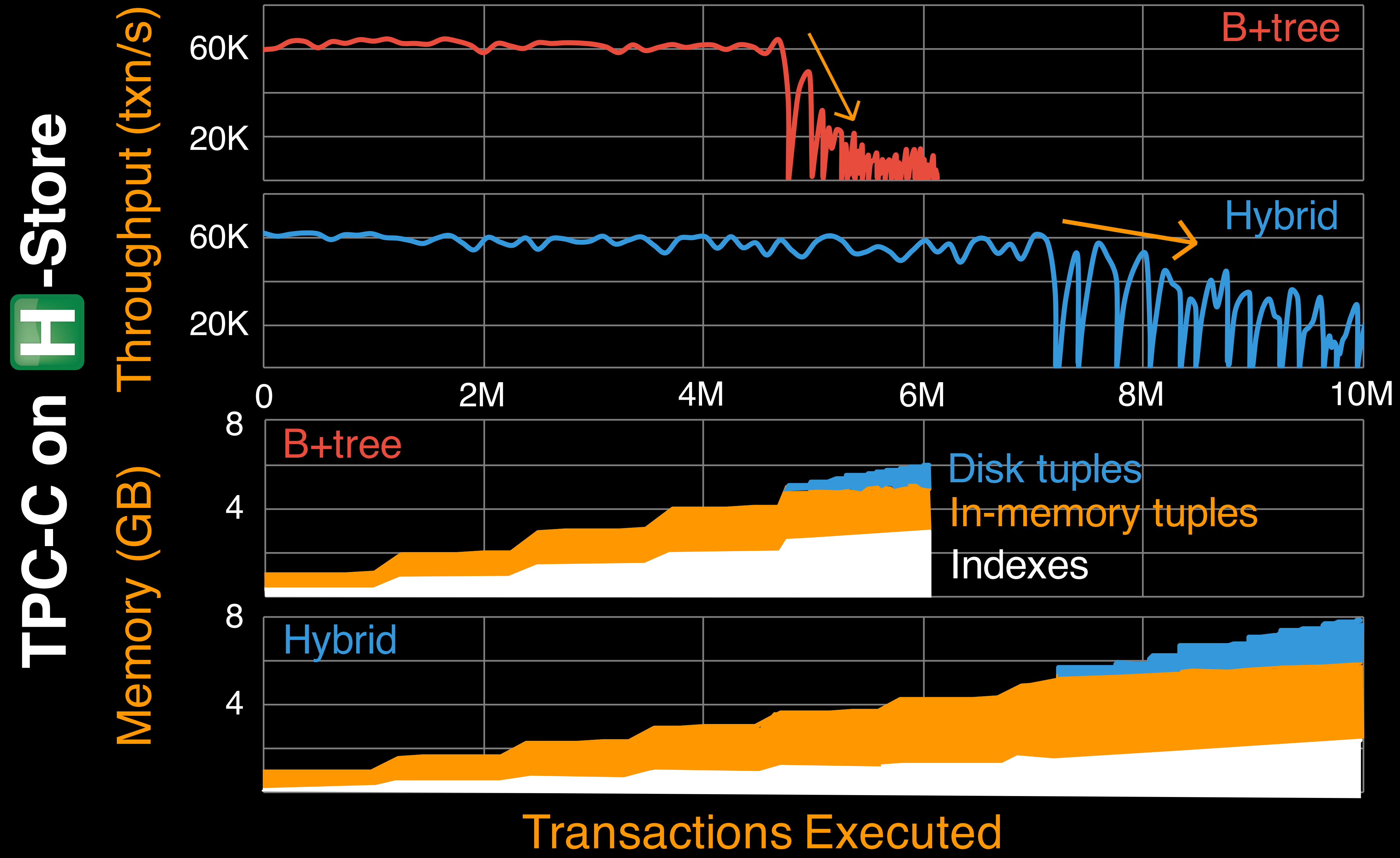


TPC-C on H-Store

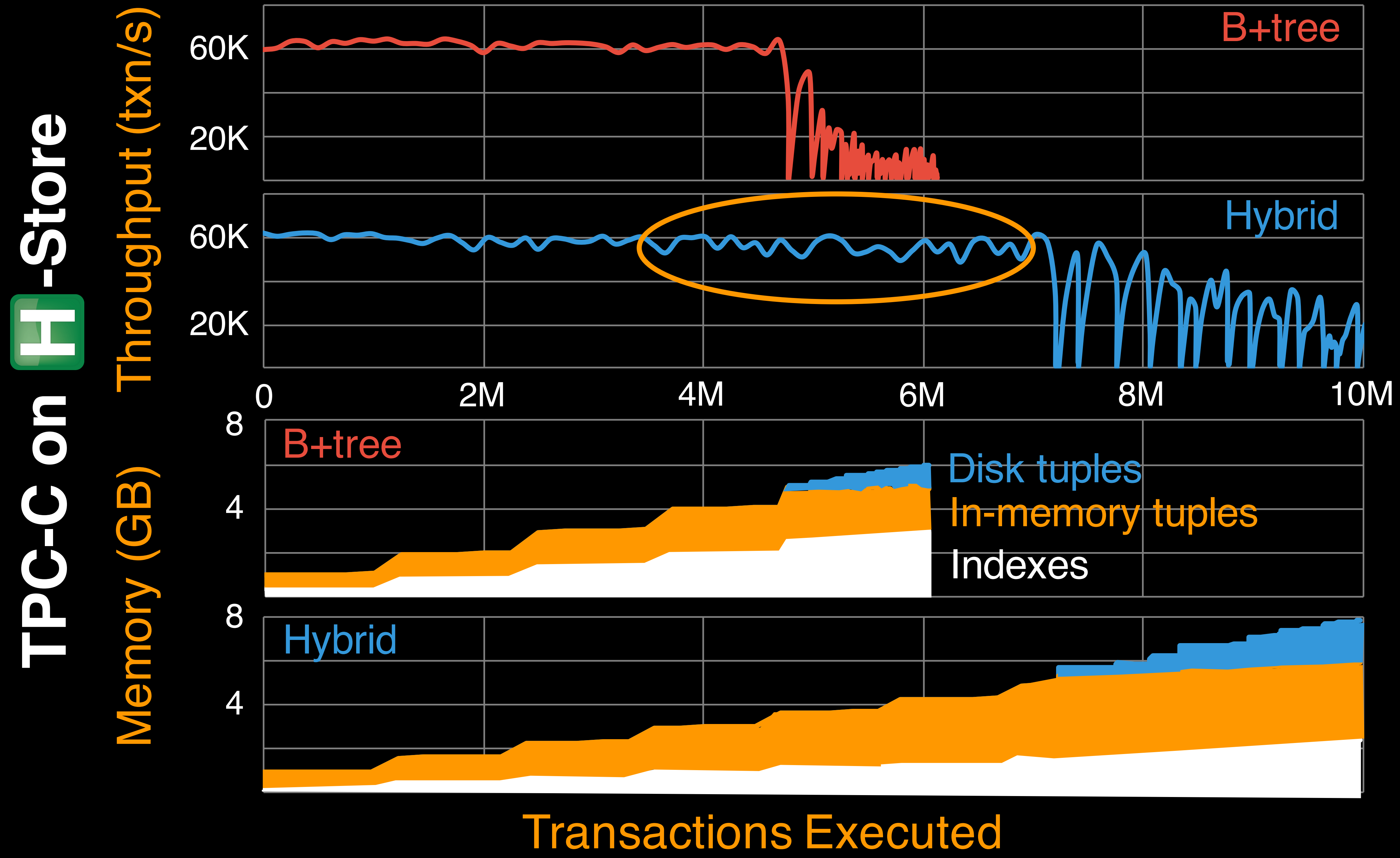




TPC-C on H-Store
Throughput (txn/s)
60K
20K
B+tree
Hybrid
0
2M
4M
6M
8M
10M
Memory (GB)
8
4
Disk tuples
In-memory tuples
Indexes
Transactions Executed

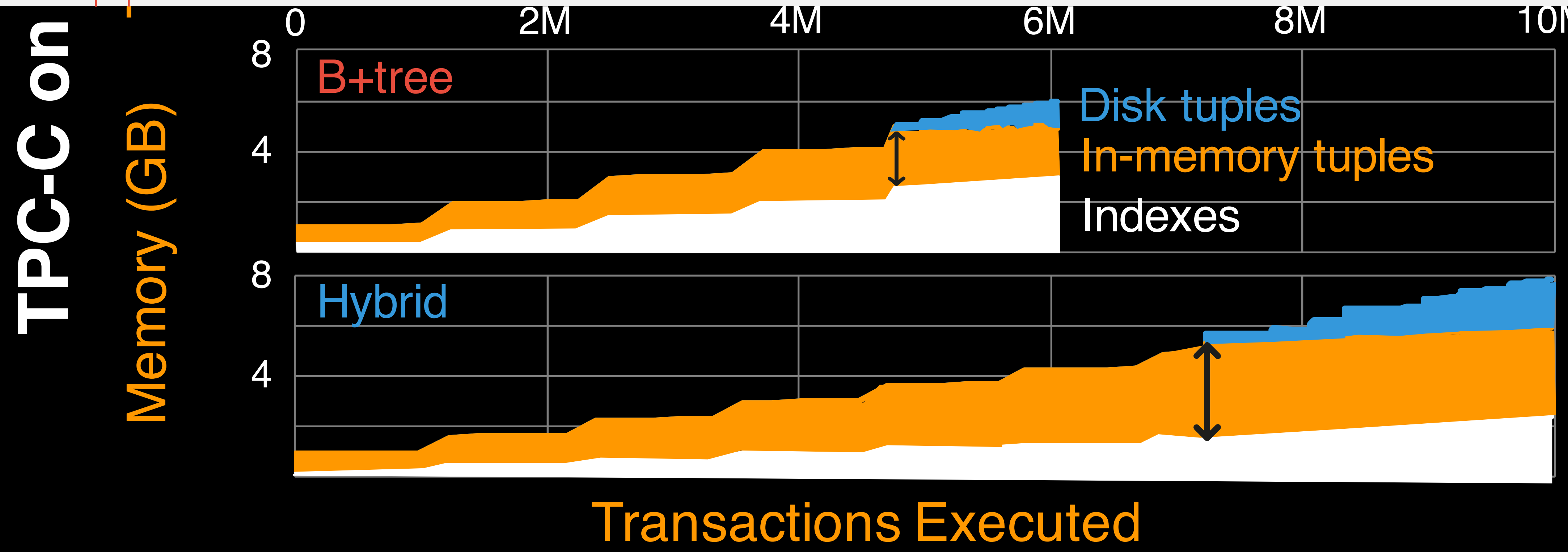


TPC-C on H-Store
Throughput (txn/s)
B+tree
Hybrid
60K
20K
0
2M
4M
6M
8M
10M
Memory (GB)
8
4
Disk tuples
In-memory tuples
Indexes
Transactions Executed


TPC-C on H-Store
Throughput (txn/s)
60K
20K
B+tree
Hybrid
0
2M
4M
6M
8M
10M
Memory (GB)
8
4
Disk tuples
In-memory tuples
Indexes
Transactions Executed

Take Away:

Memory saved by indexes → Larger working set in memory → Higher throughput

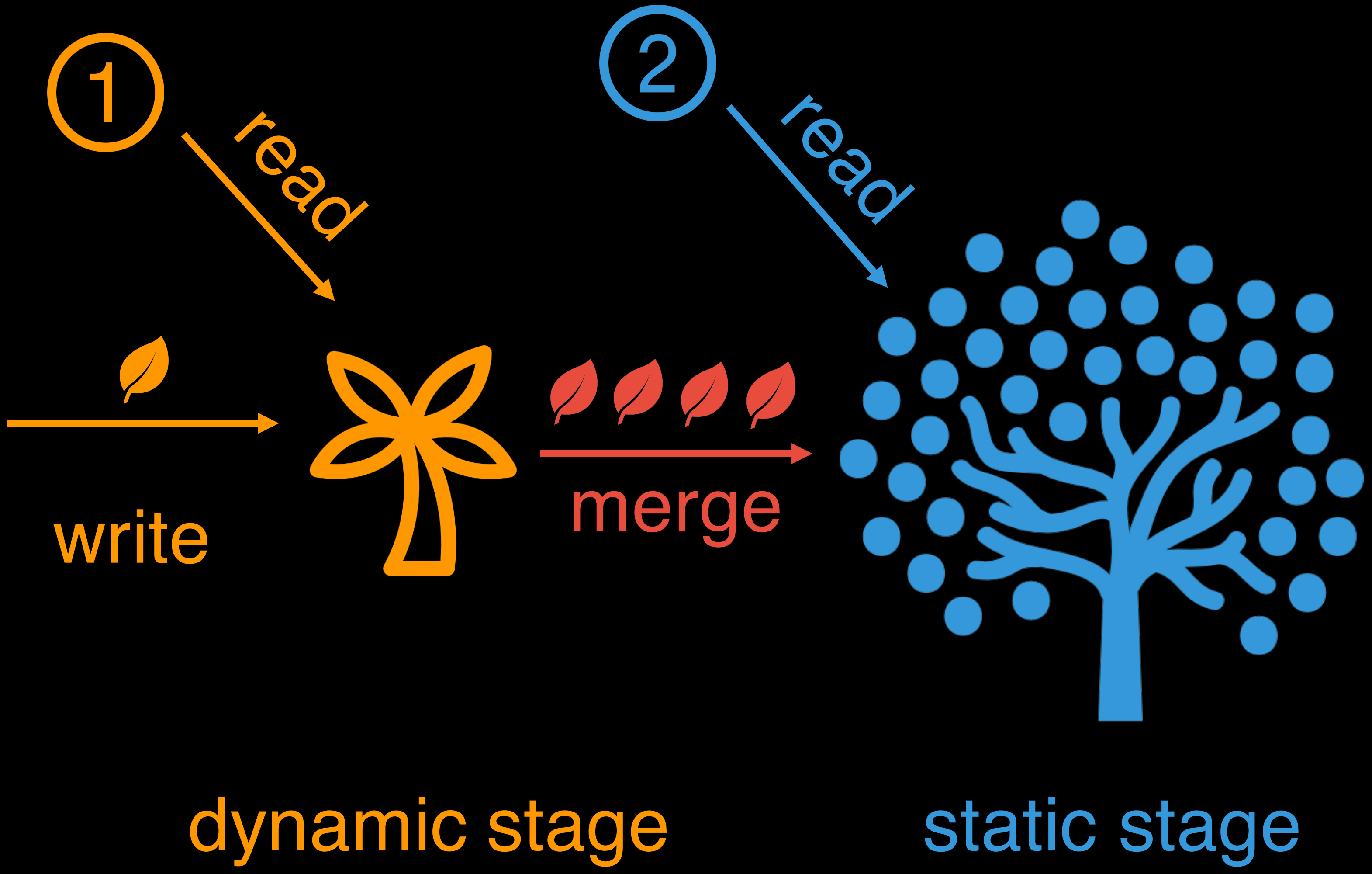


Part I Recap

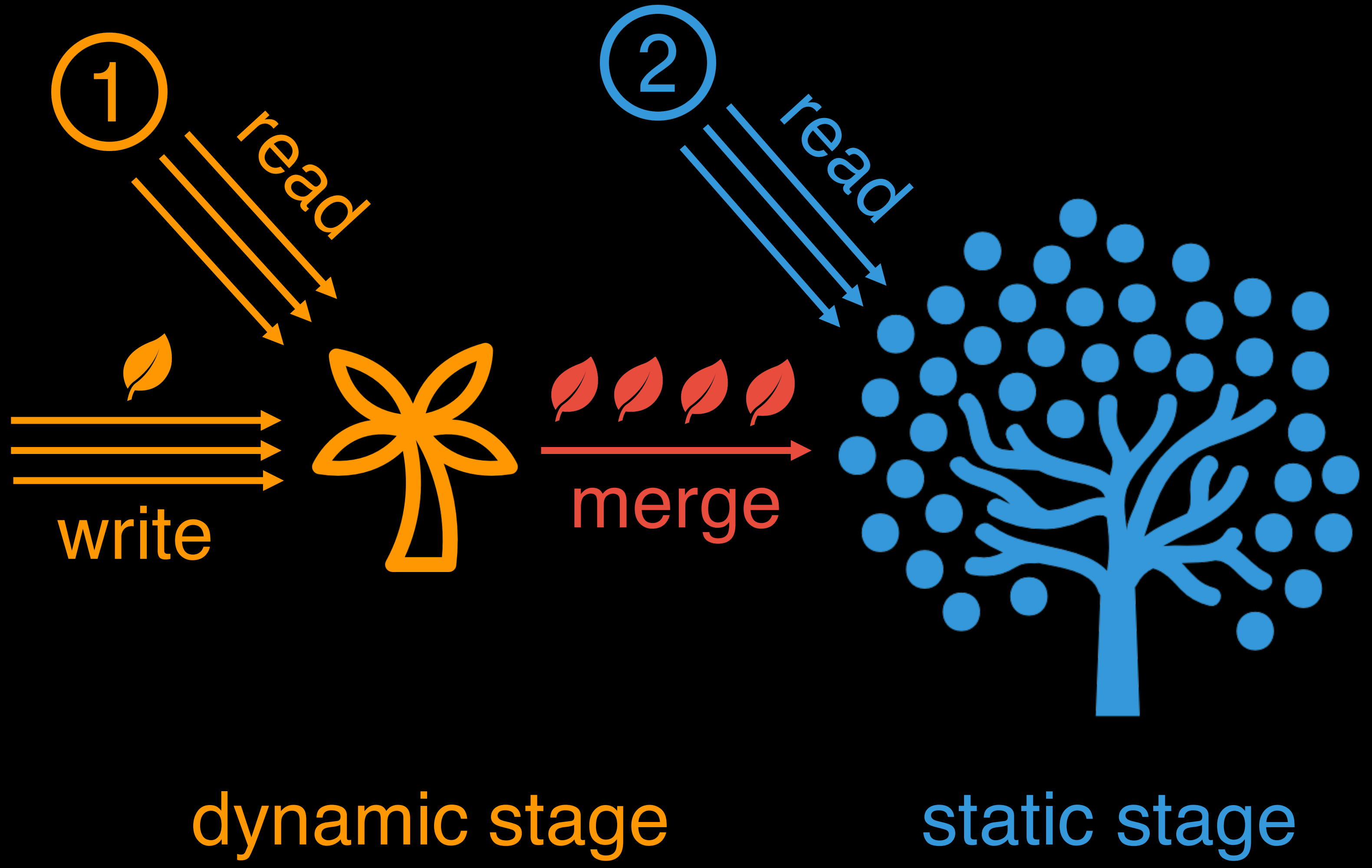
1. The hybrid index architecture — **GENERAL**
2. The Dual-Stage Transformation — **PRACTICAL**
3. Applied to 4 index structures — **USEFUL**
   - B+tree
   - Skip List
   - Masstree
   - Adaptive Radix Tree (ART)

Part II
Concurrent hybrid indexes with non-blocking merge

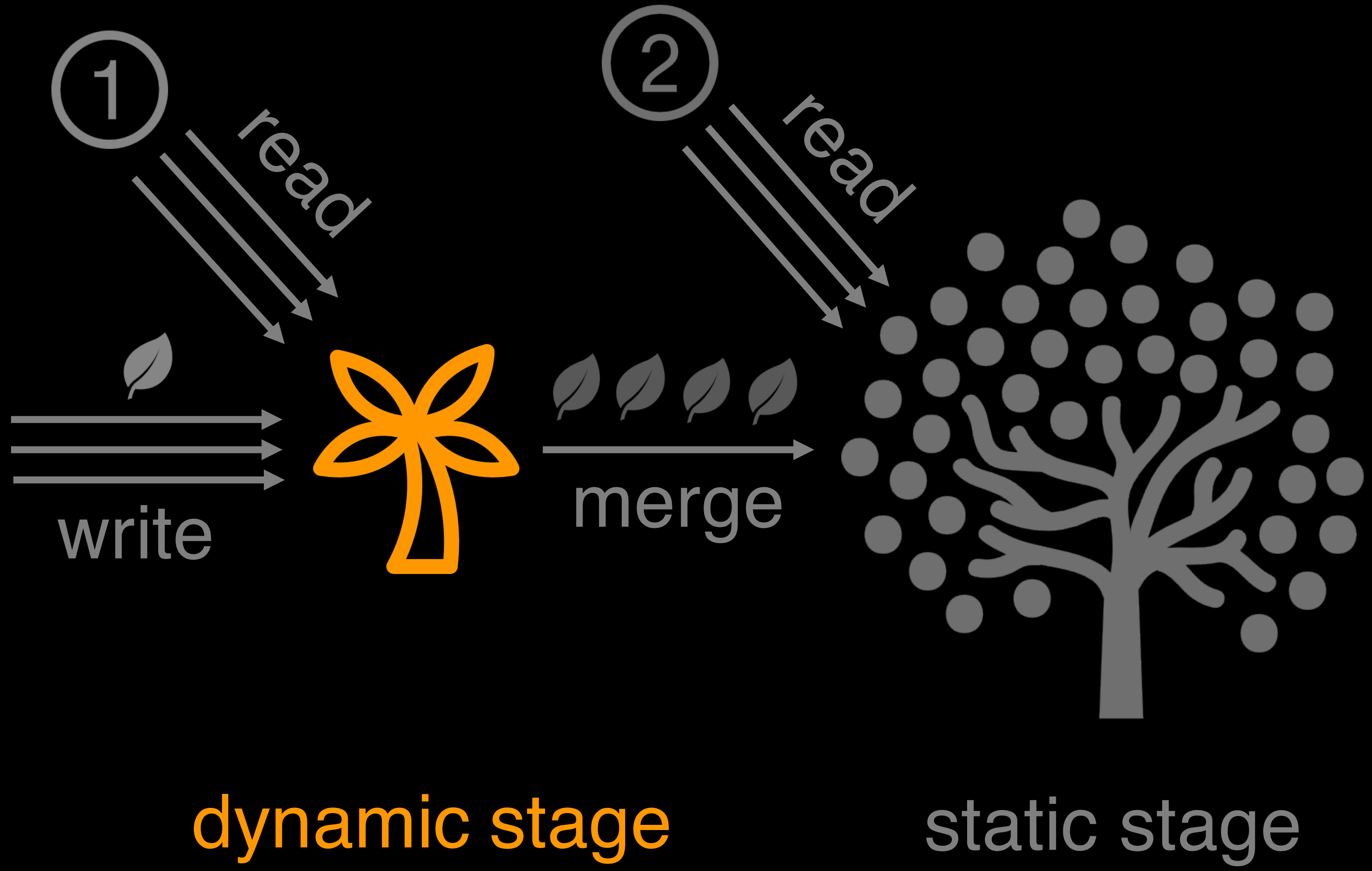
Building Concurrent Hybrid Index?



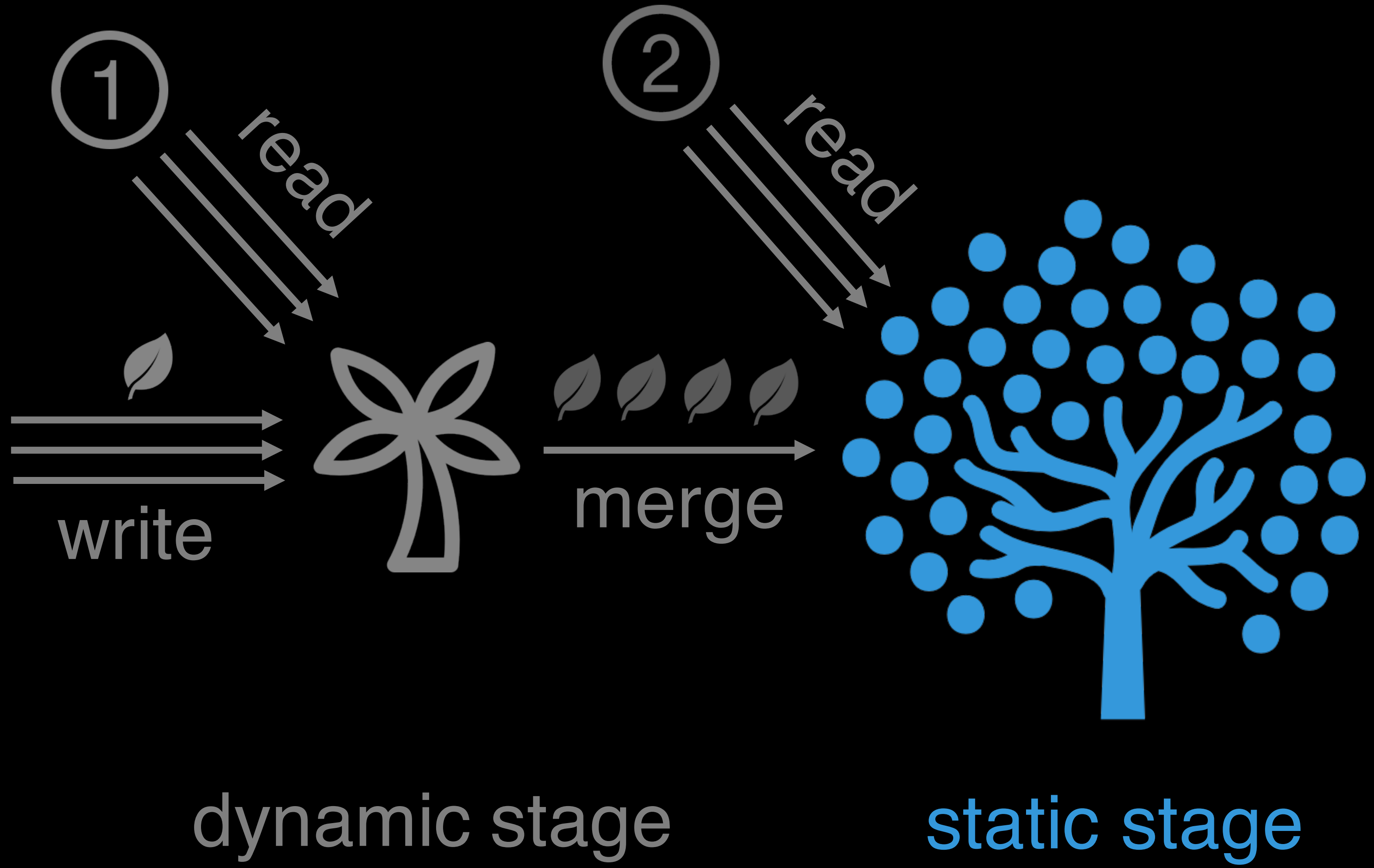
Building Concurrent Hybrid Index?



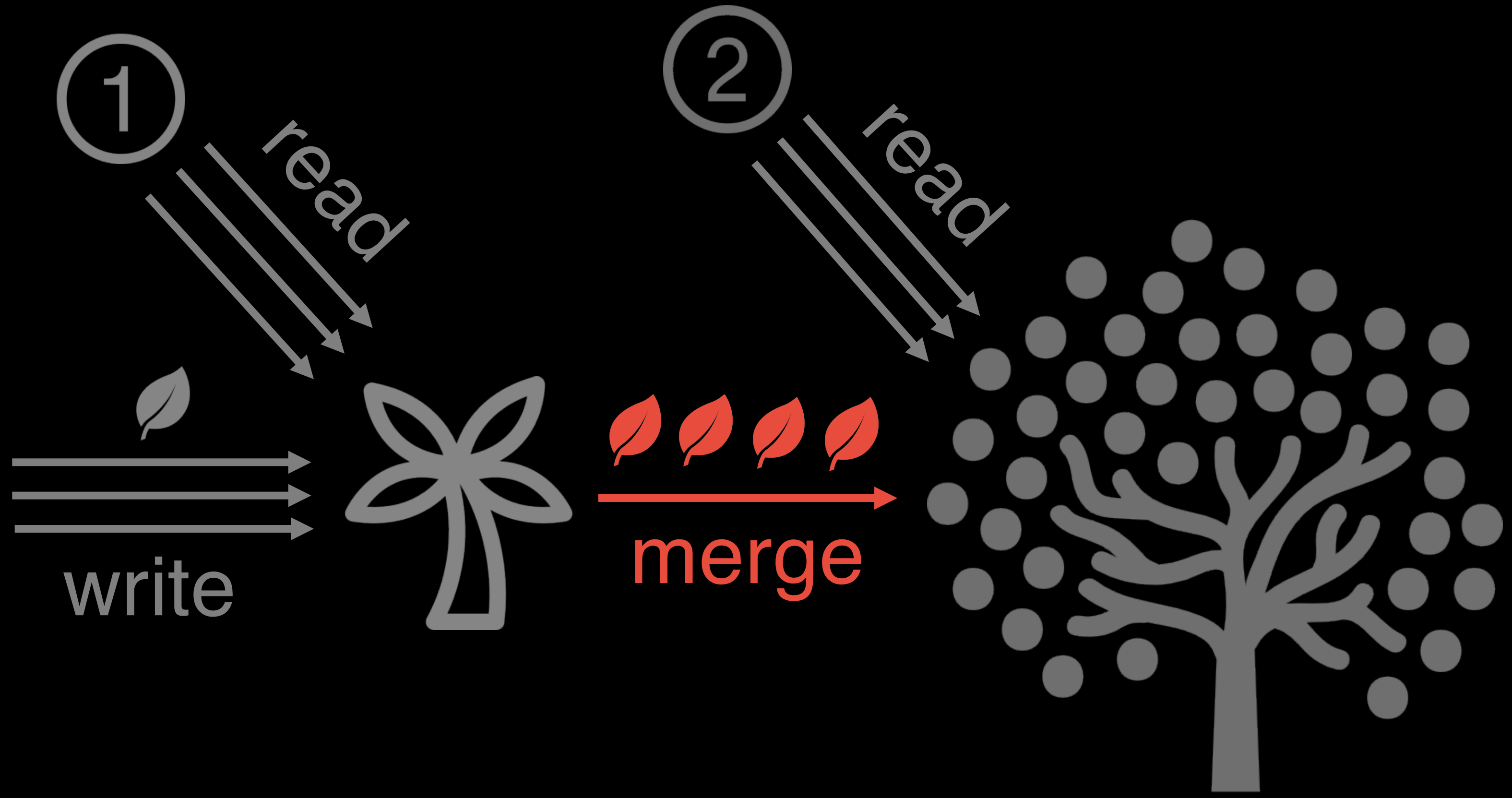
Use concurrent data structures for dynamic-stage



Static-stage is perfectly concurrent by default



Challenge: efficient non-blocking merge algorithm



dynamic stage

static stage

Merge Algorithm Requirements

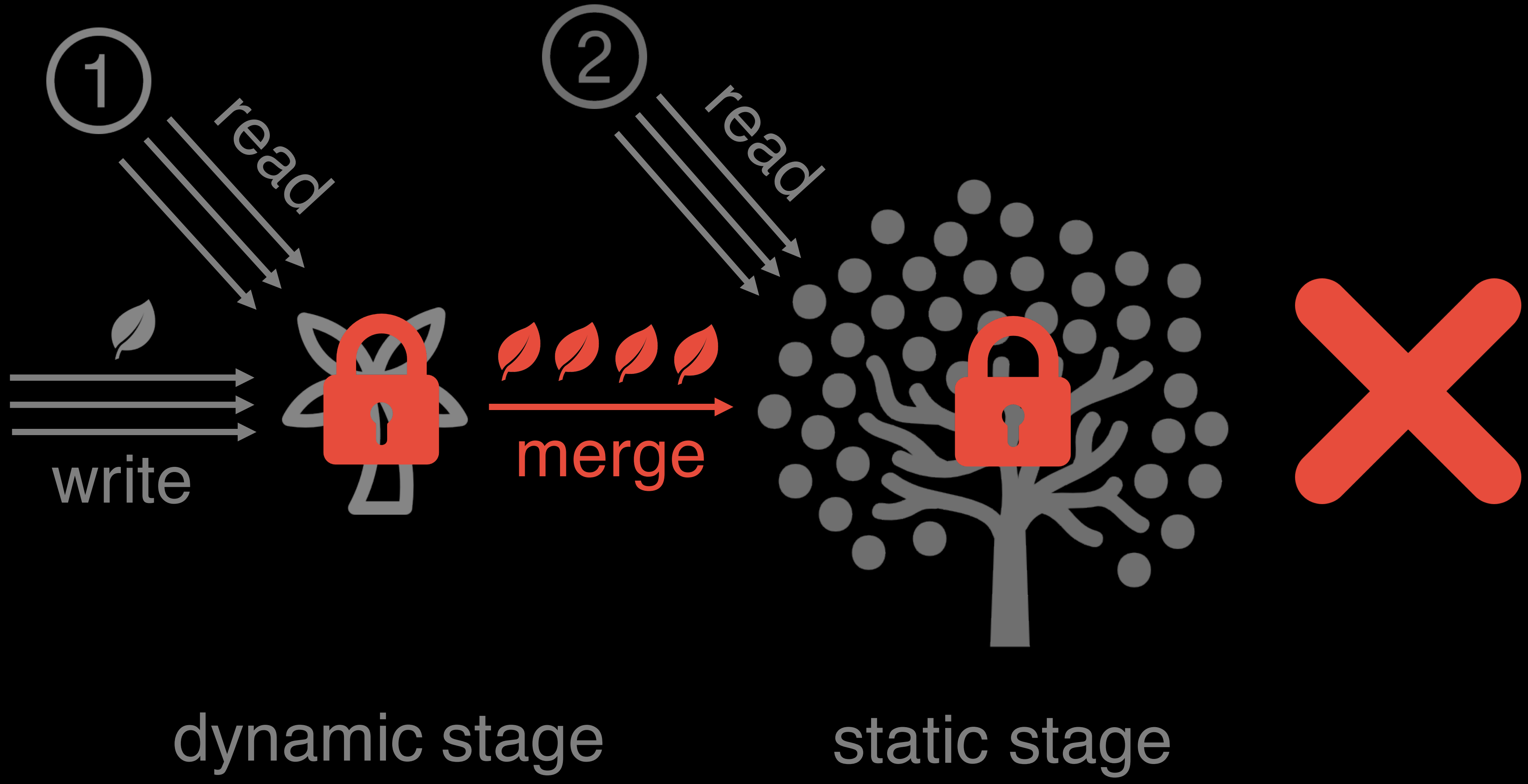
1 Non-blocking

- All existing items are accessible during merge
- New items can still enter

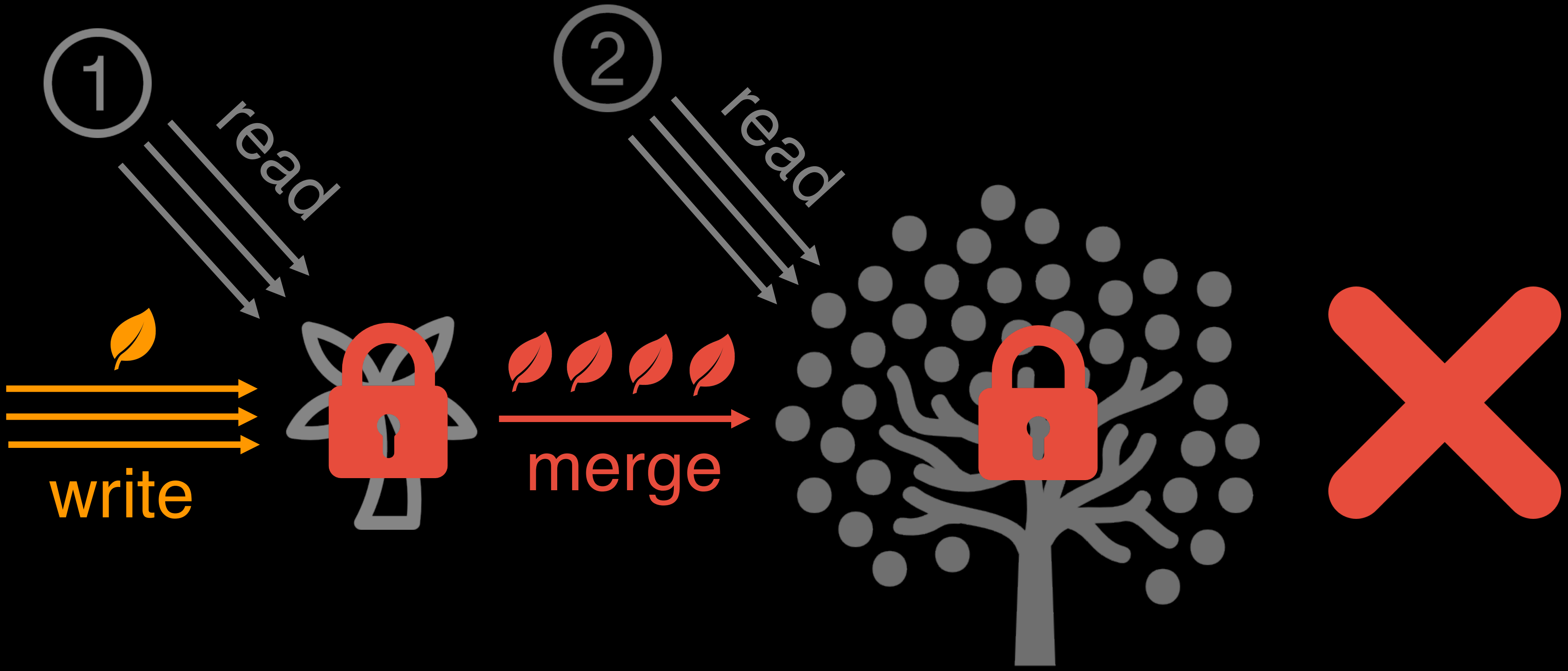
2 Efficient

- Fast
- Bounded temporary memory use

Naïve Solution 1: Coarse-grained Locking



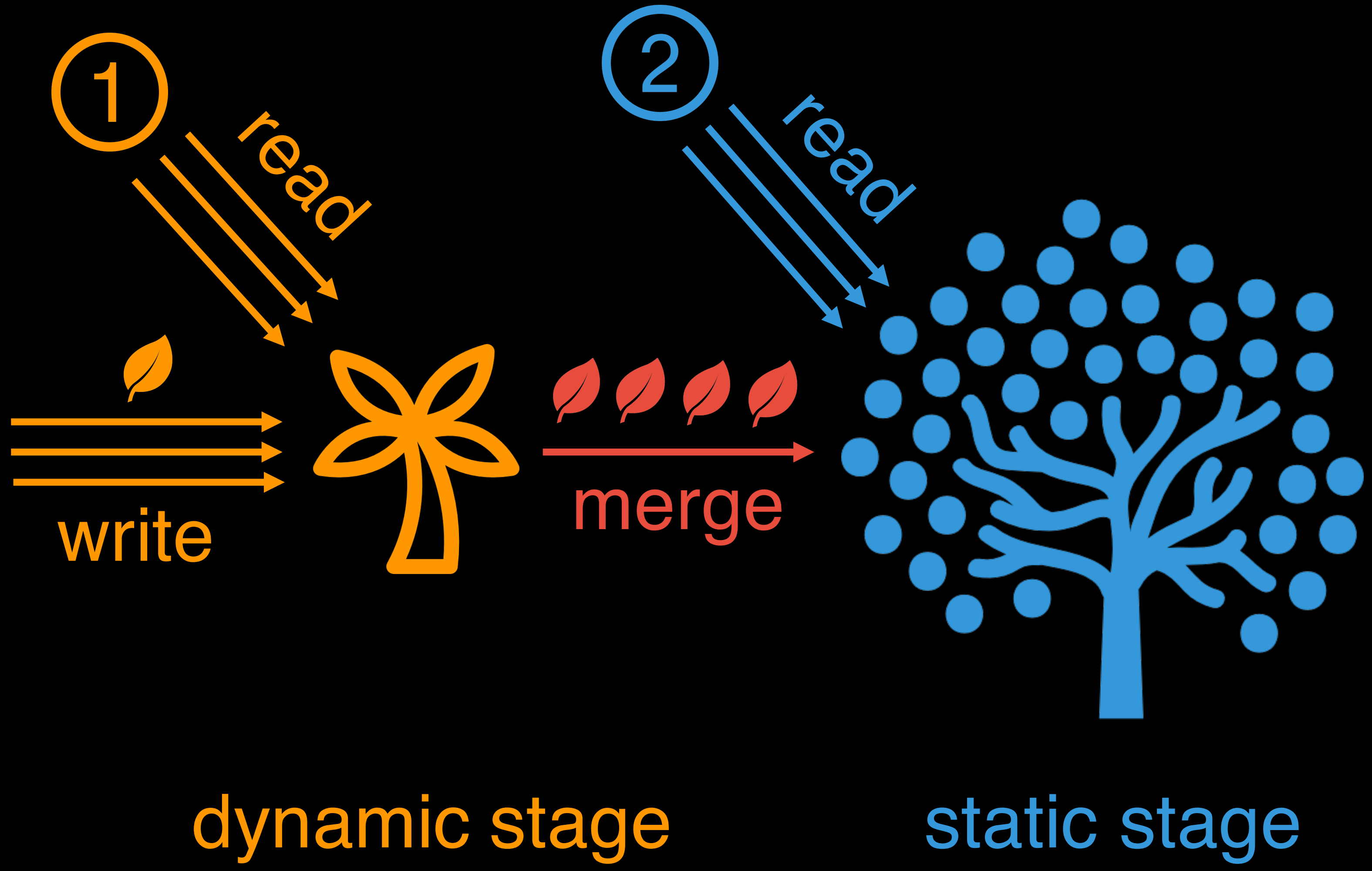
Naïve Solution 1: Coarse-grained Locking



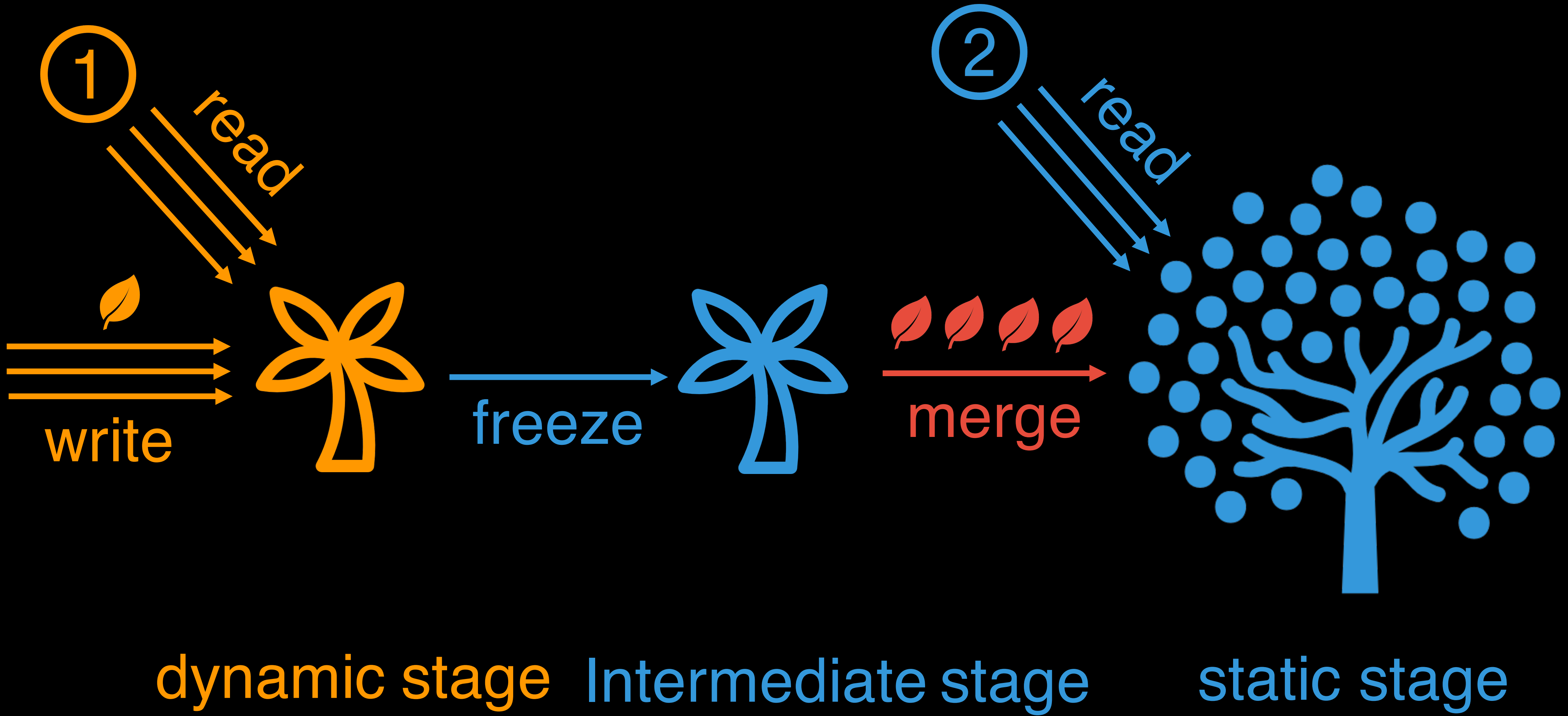
dynamic stage

static stage

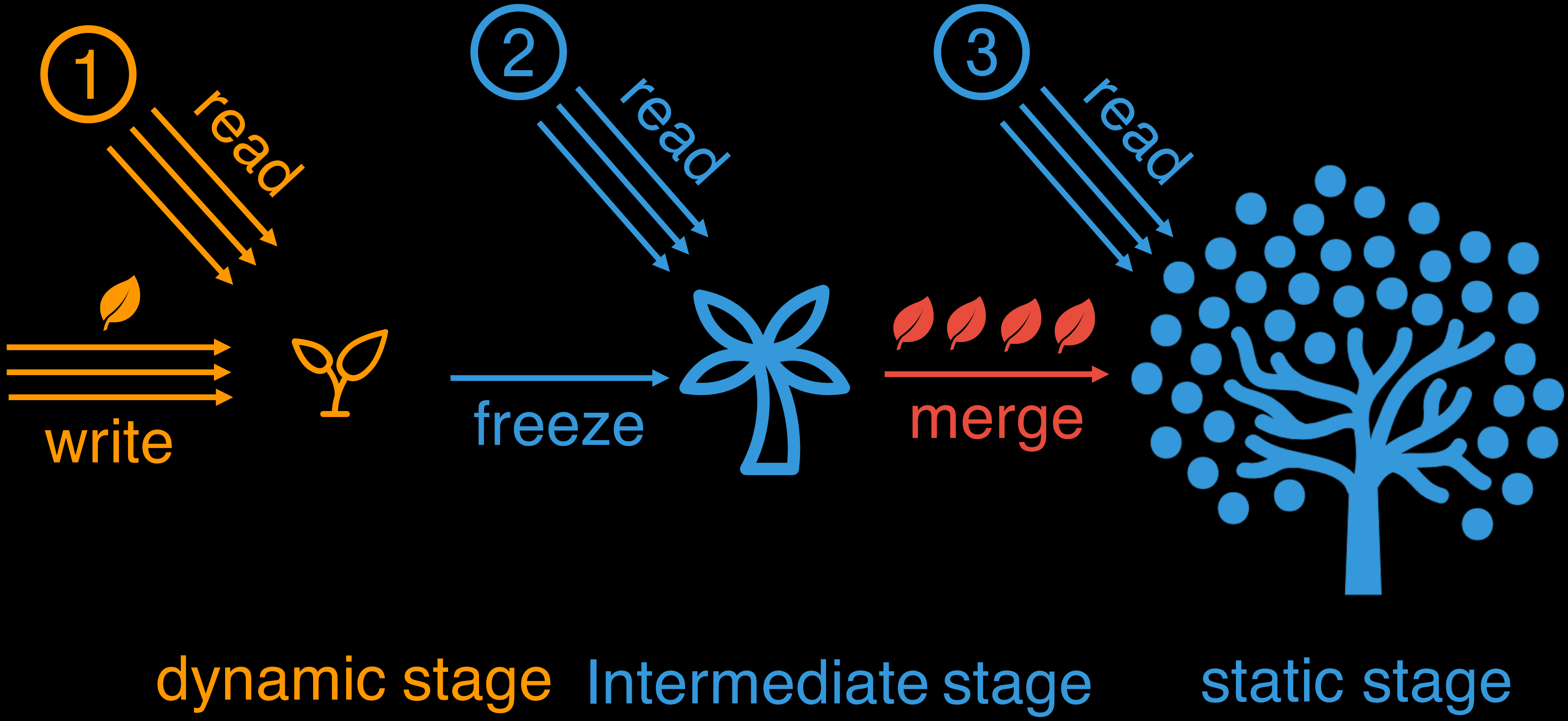
The intermediate stage unblocks write traffic



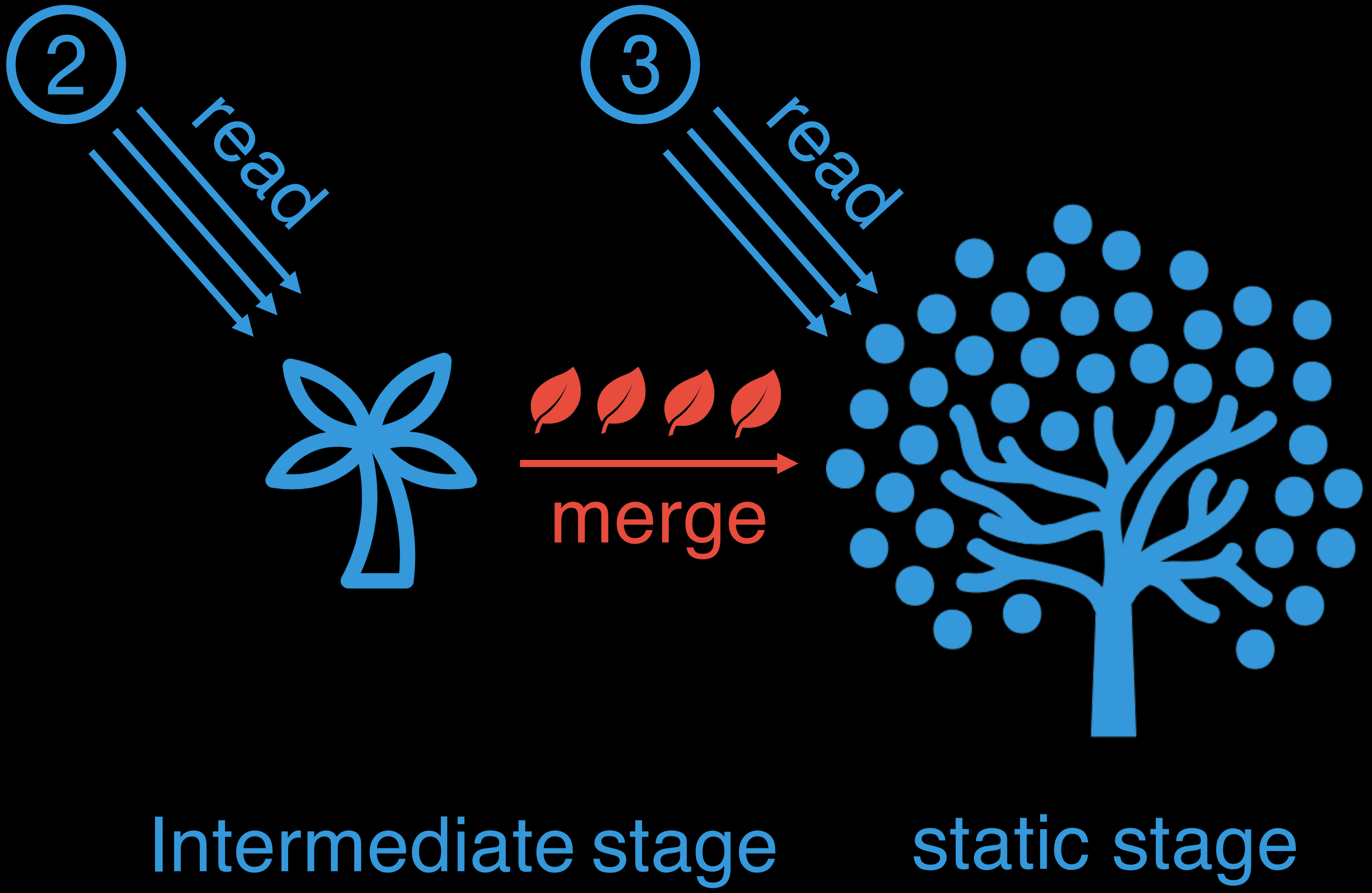
The intermediate stage unblocks write traffic



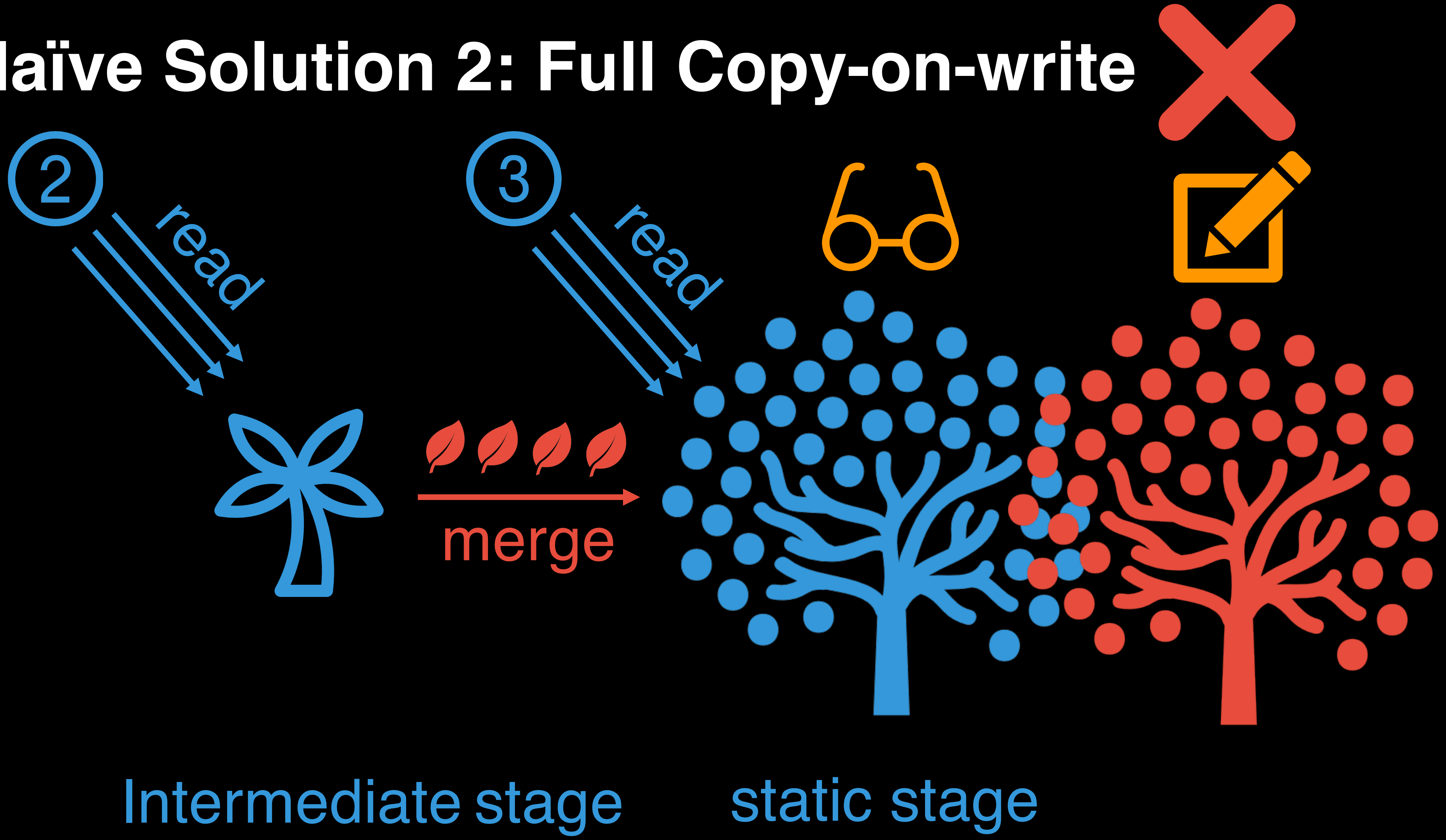
The intermediate stage unblocks write traffic



How do we unblock reads during merge?



Naïve Solution 2: Full Copy-on-write

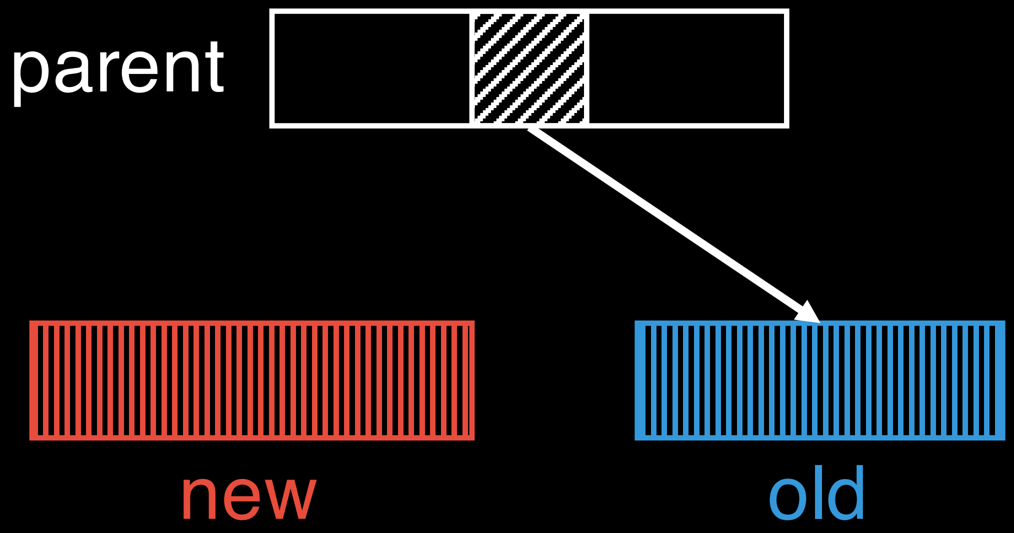


Key Observation

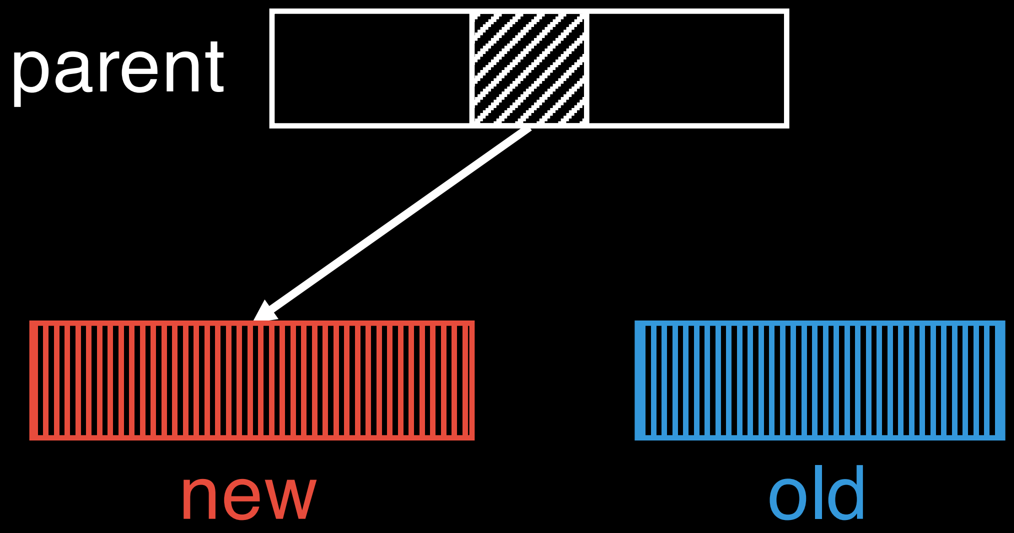
Merged-in items in the static-stage will NOT be accessed until the intermediate-stage is deleted

⇒ Merge Incrementally!

Our Solution: Incremental Copy-on-write with Rapid GC

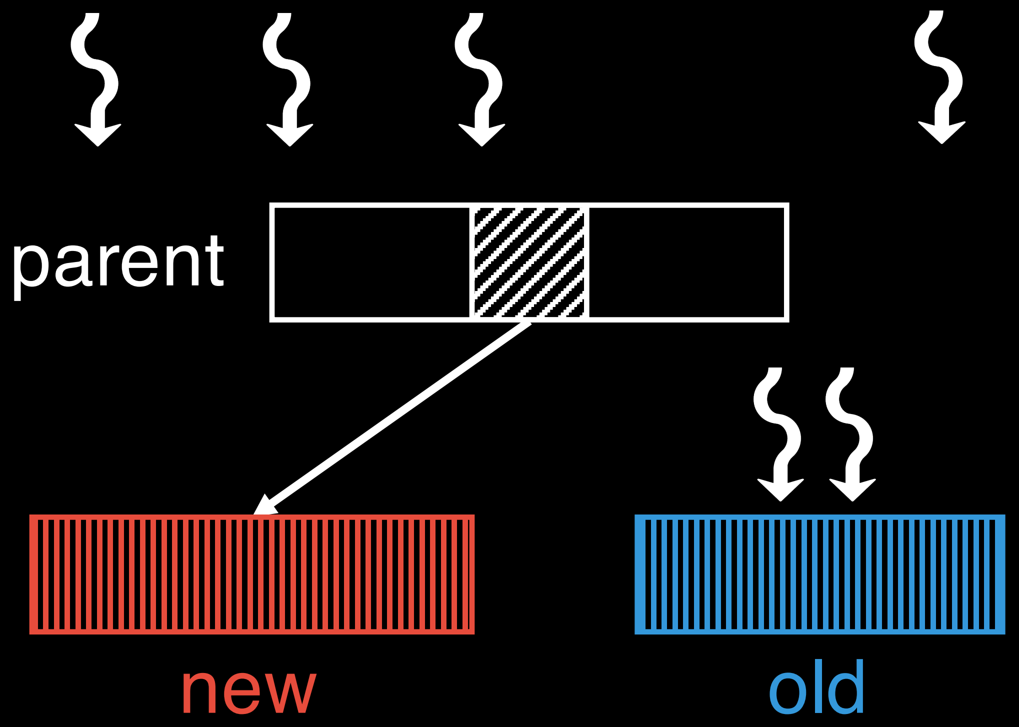


Our Solution: Incremental Copy-on-write with Rapid GC



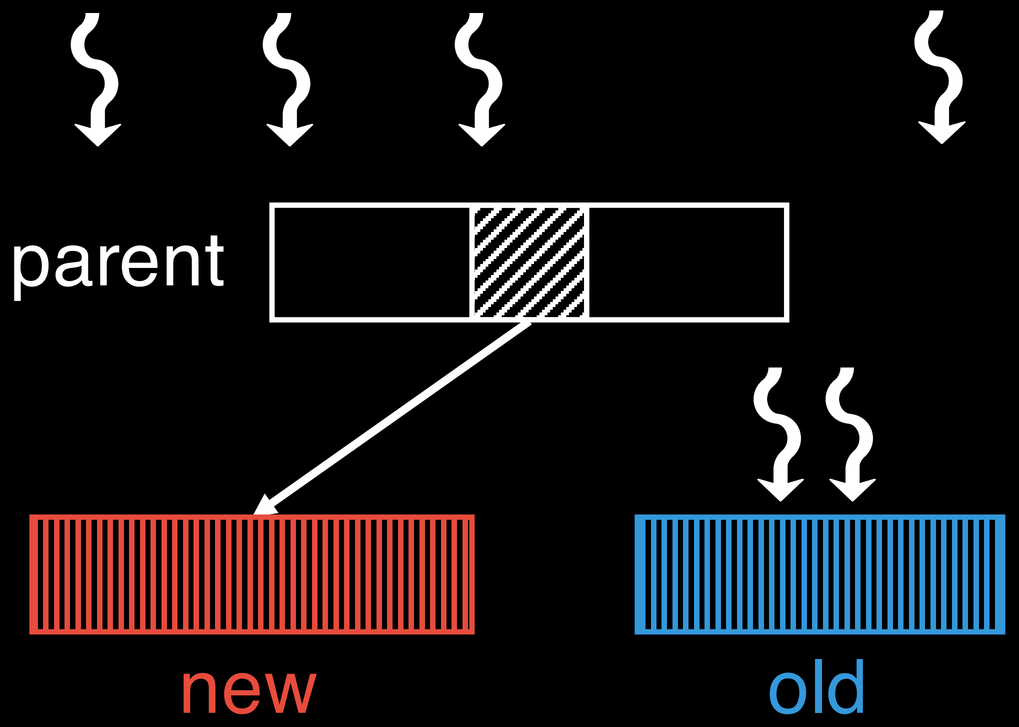
When can we safely reclaim the garbage?

Our Solution: Incremental Copy-on-write with Rapid GC



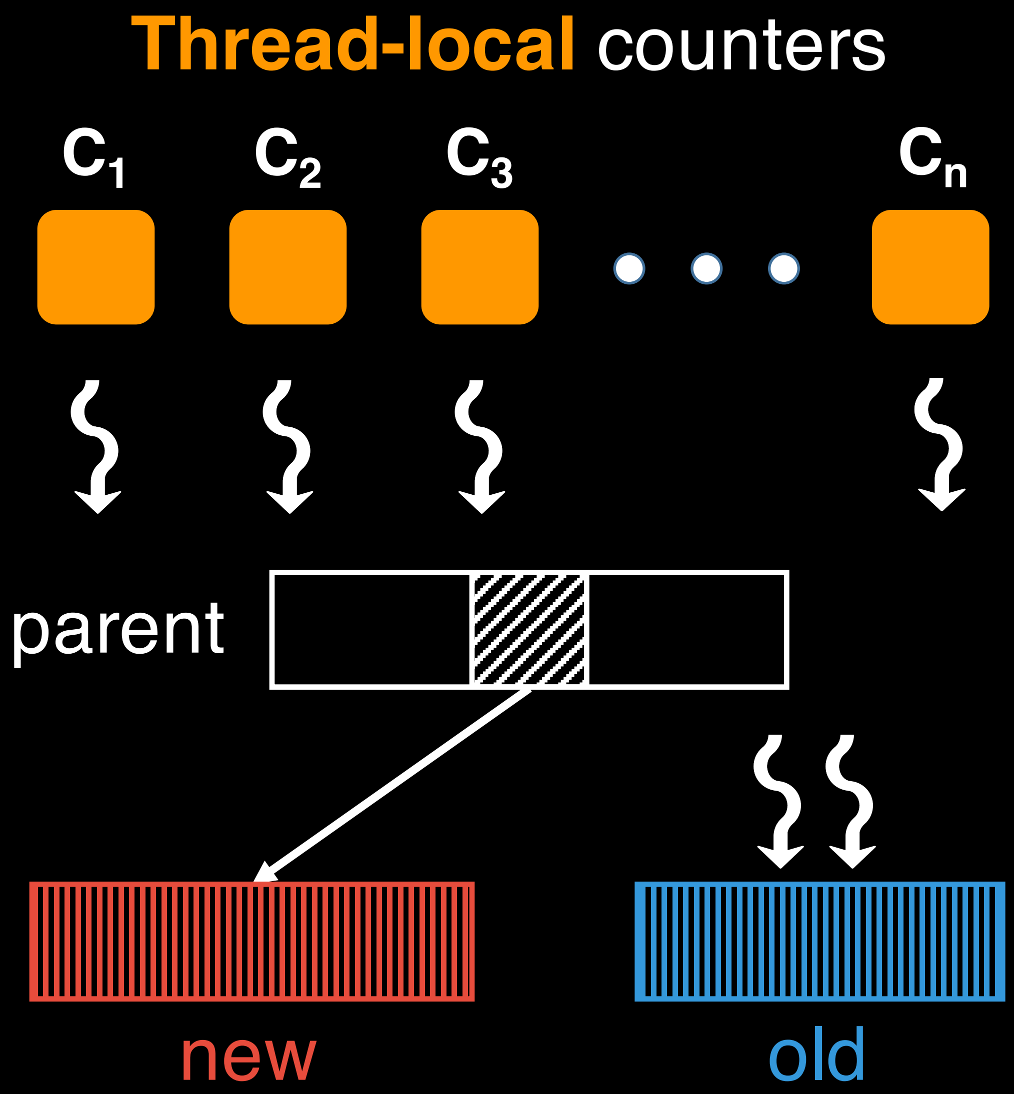
When can we safely reclaim the garbage?

Our Solution: Incremental Copy-on-write with Rapid GC



When no thread still holds a reference to it!

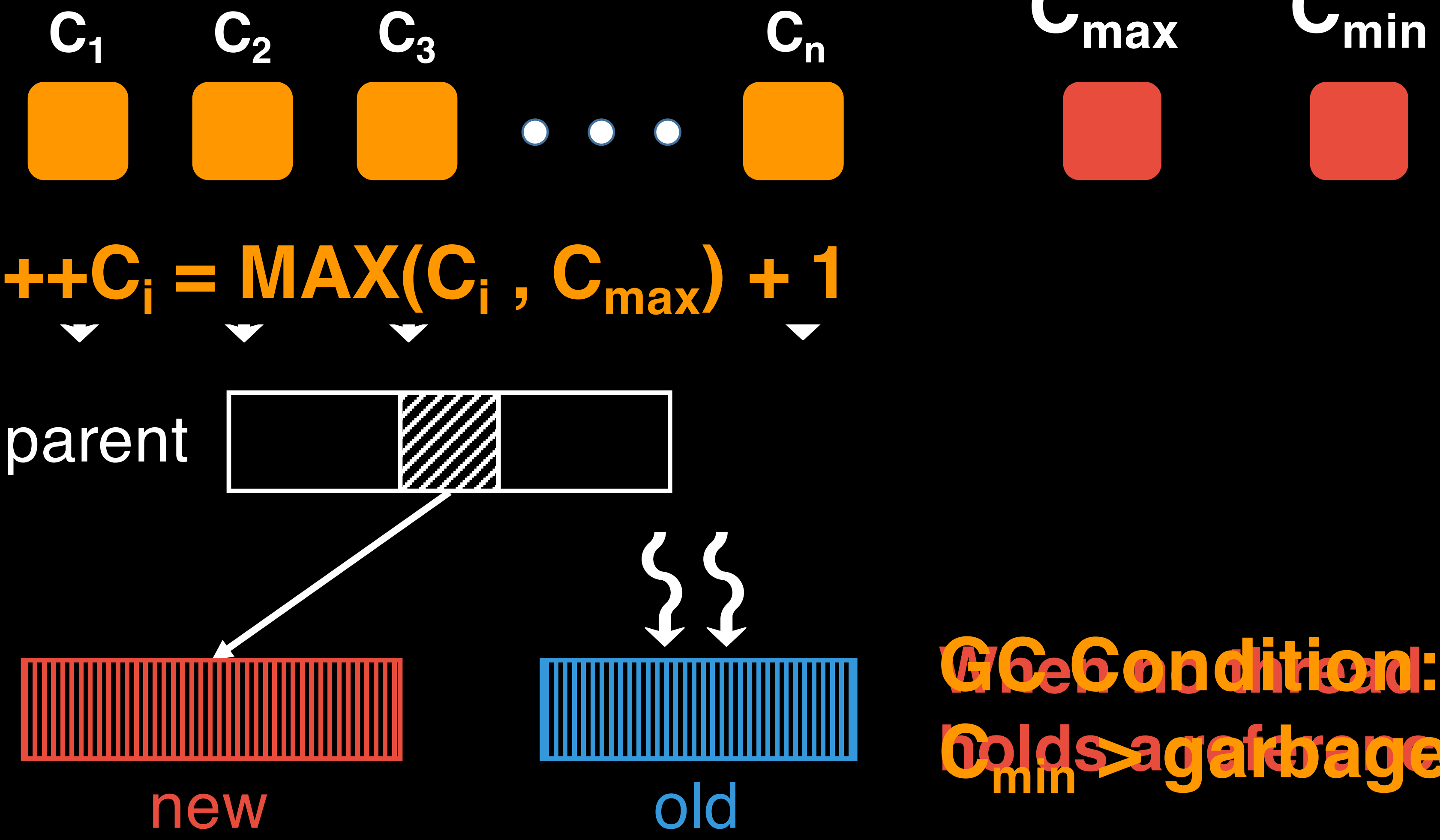
Our Solution: Incremental Copy-on-write with Rapid GC



When no thread still holds a reference to it!

Our Solution: Incremental Copy-on-write with Rapid GC

Thread-local counters



GC Condition:
C_{min} > garbage tag

When no thread still
holds a reference to it!

A Quick Recap of the Merge Algorithm

1. The intermediate stage separates writes from the merge process

2. The incremental merge algorithm with rapid GC is non-blocking and space-efficient

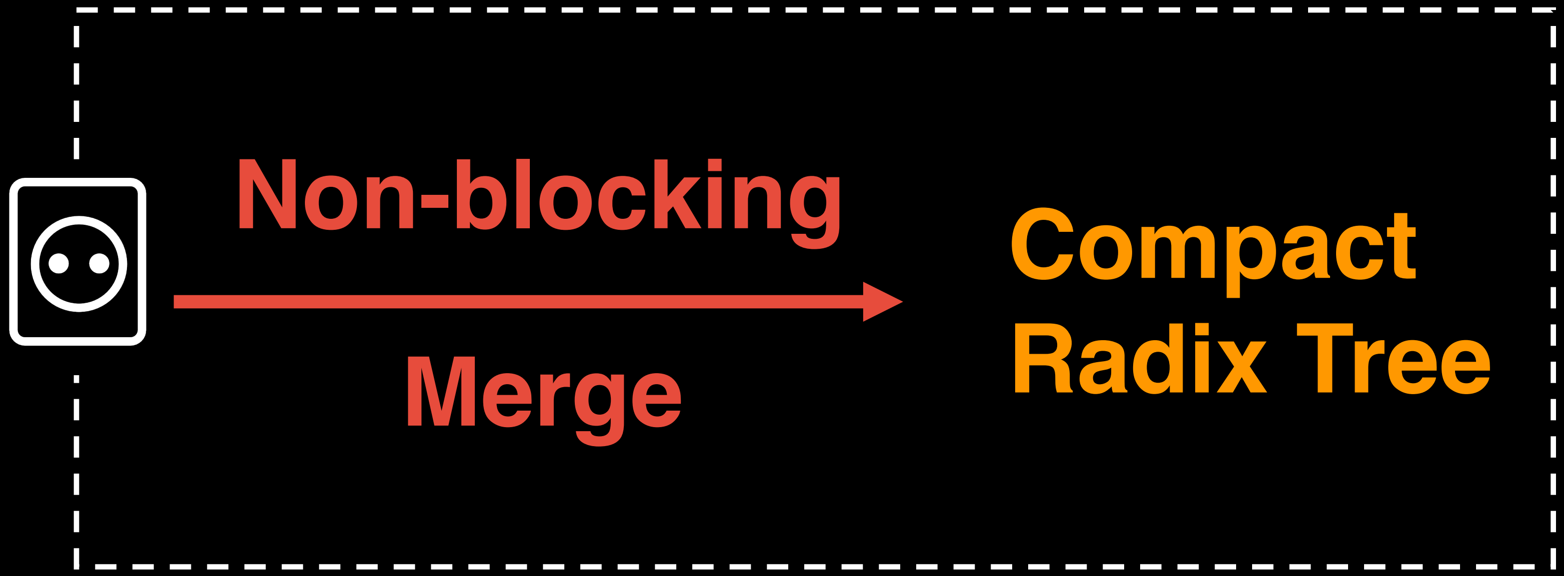
What we are building now

Non-blocking

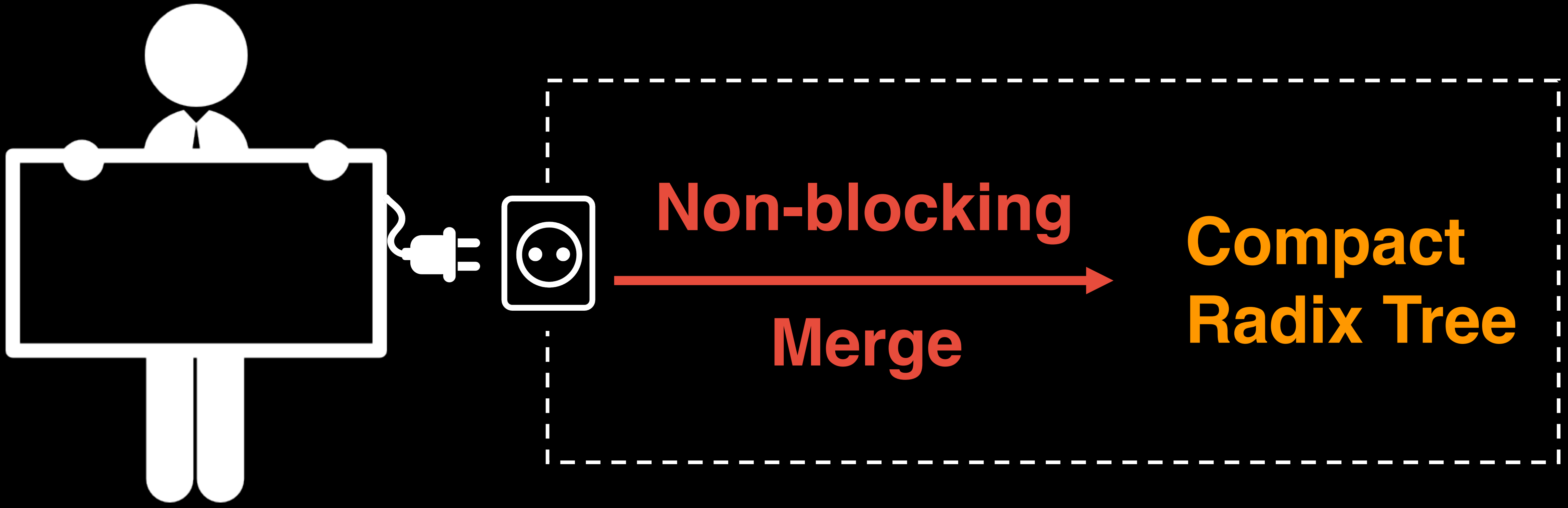
Merge

Compact
Radix Tree

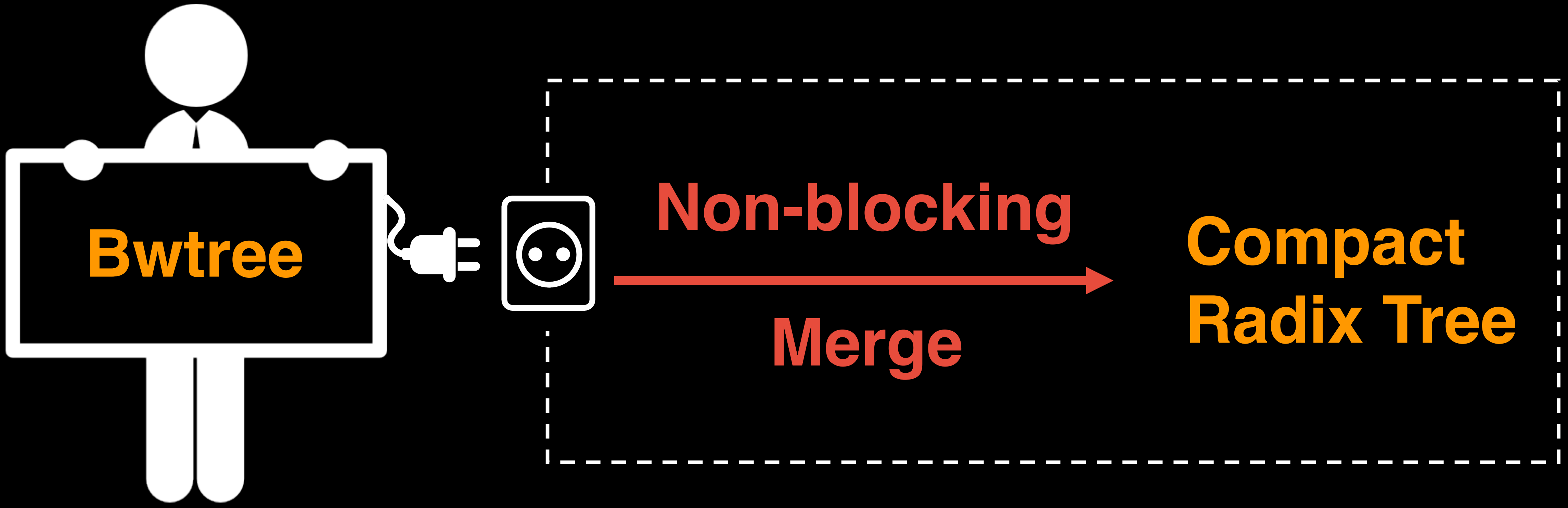
What we are building now



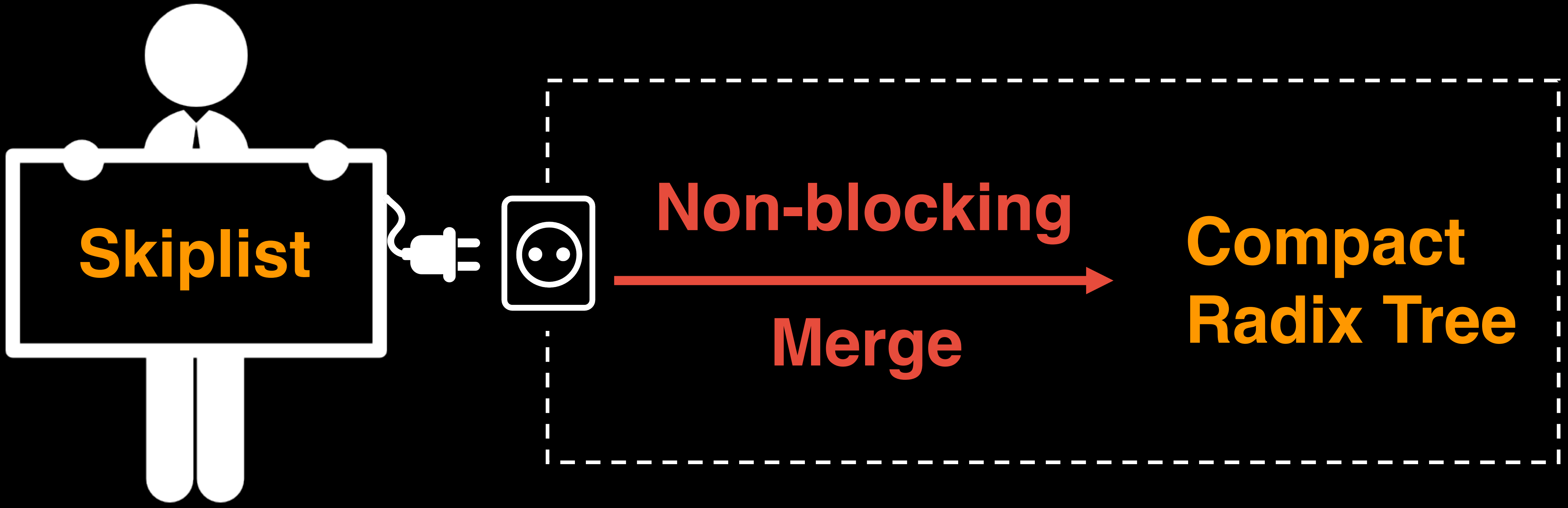
What we are building now



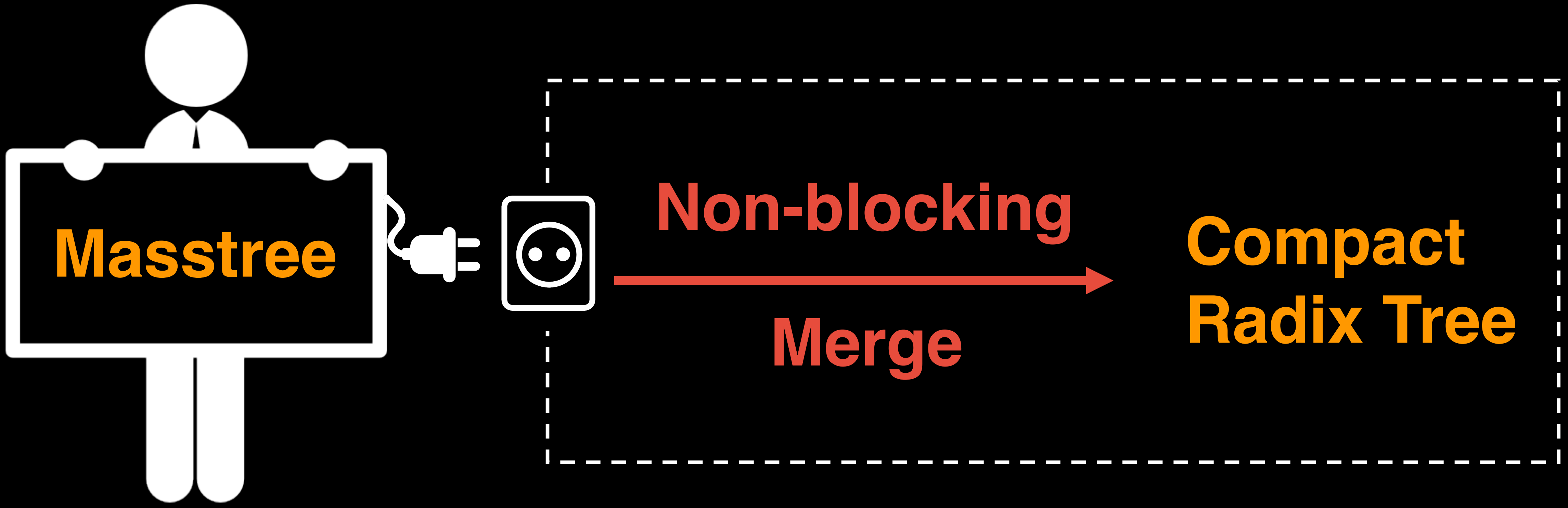
What we are building now



What we are building now



What we are building now



Part III
Super-compact
static-stage

Go “crazy” on space-efficiency

Succinct Data Structures

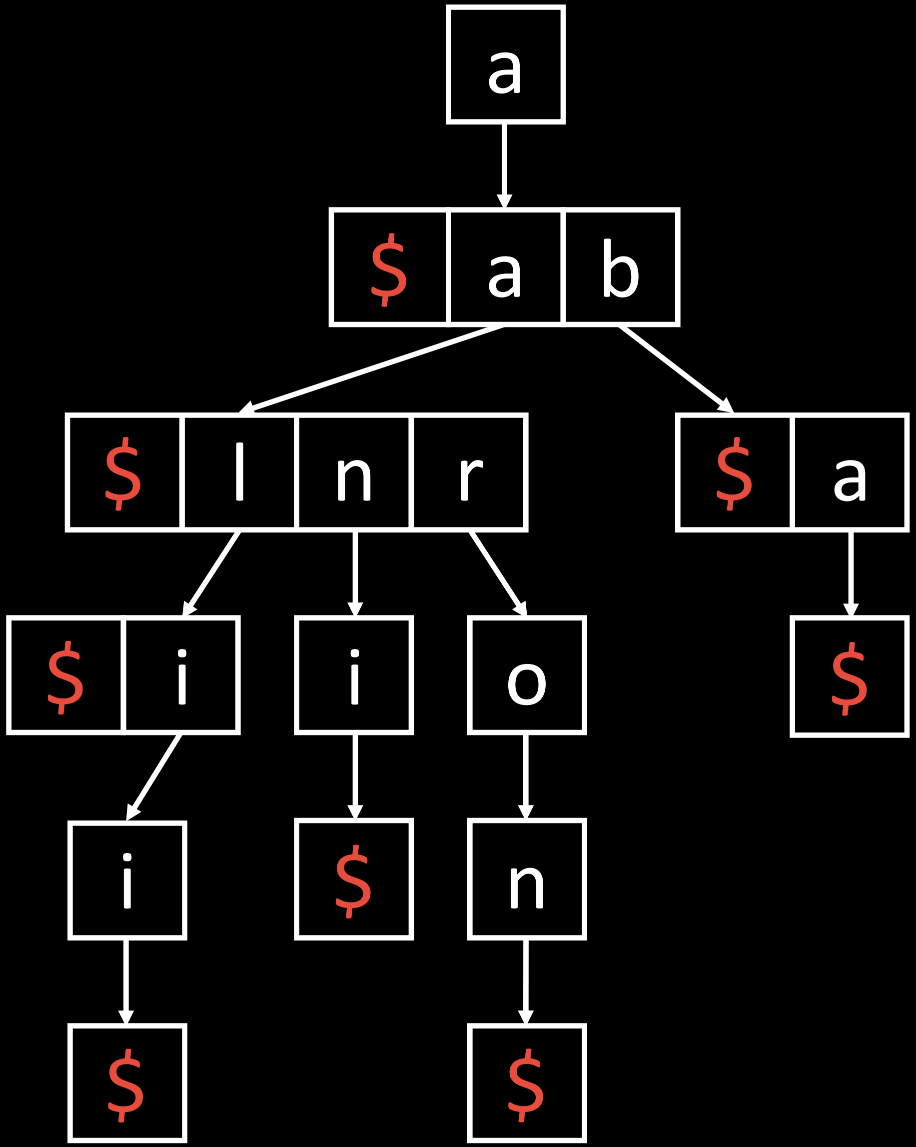
- $Z + o(Z)$, where Z is the information-theoretic lower bound
- Still allow for efficient query operations

100011010000101…

$rank_1(x)$ = # of 1’s up to position x

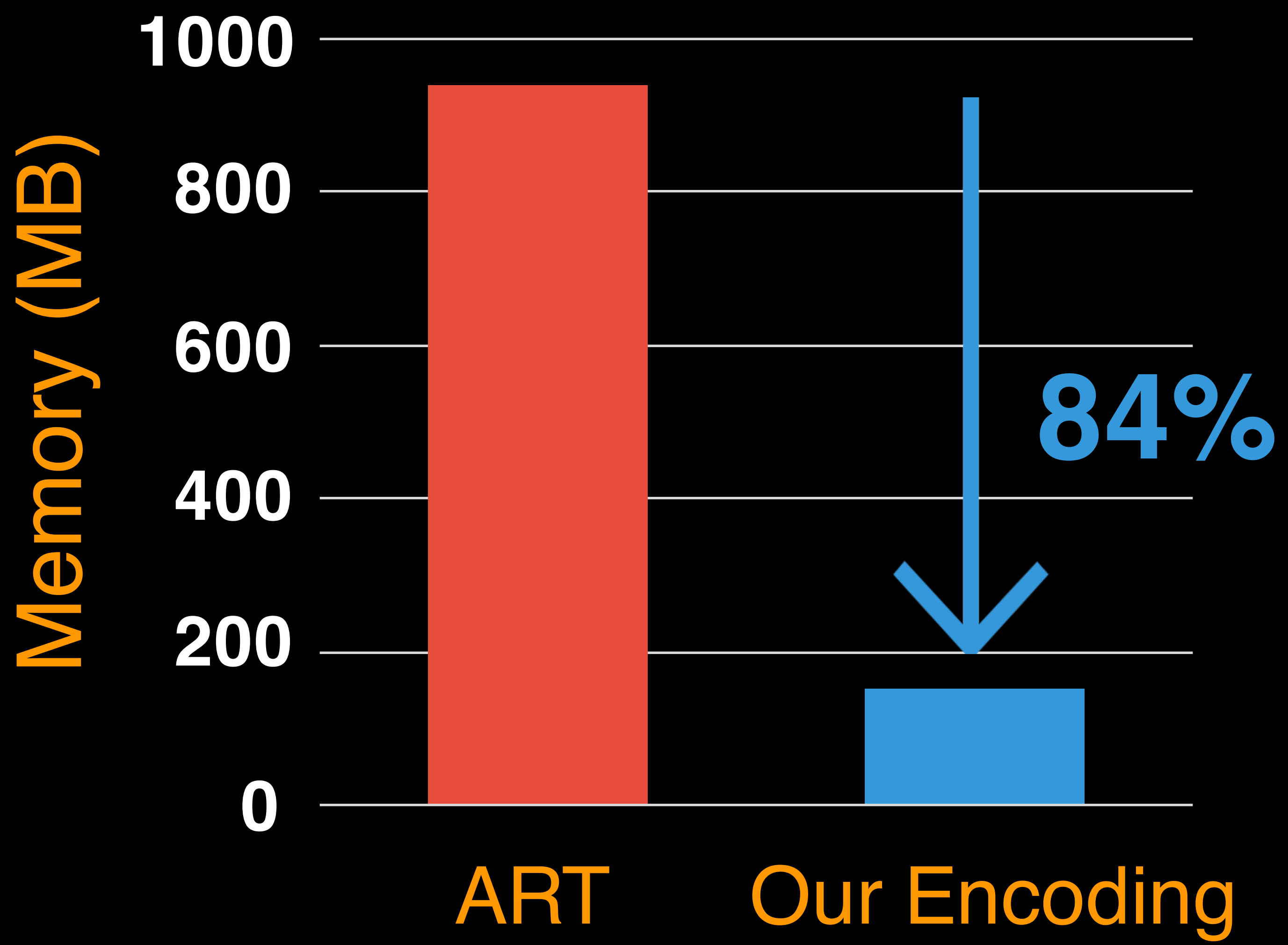
$select_1(x)$ = position of the x-occurrence of 1

Encoding Radix Tree



a	0	10
$ab	100	1000
lnra	100010	10000100
iio	10001	100101010
i$n	010	101010
$$	11	1010

Memory Savings with the New Encoding



50M email keys with average length = **20** bytes

The Takeaway Message

Hybrid indexes can save the precious memory resources with minimum performance penalty.

Toll-Free Hotline:

1-844-88-CMUDB

Back-up Slides

Latency (ms)

	B+tree	Hybrid
50%	10	10
99%	50	52
MAX	115	611

YCSB-based Microbenchmark Evaluation

Workload: insert, read/update(50/50)

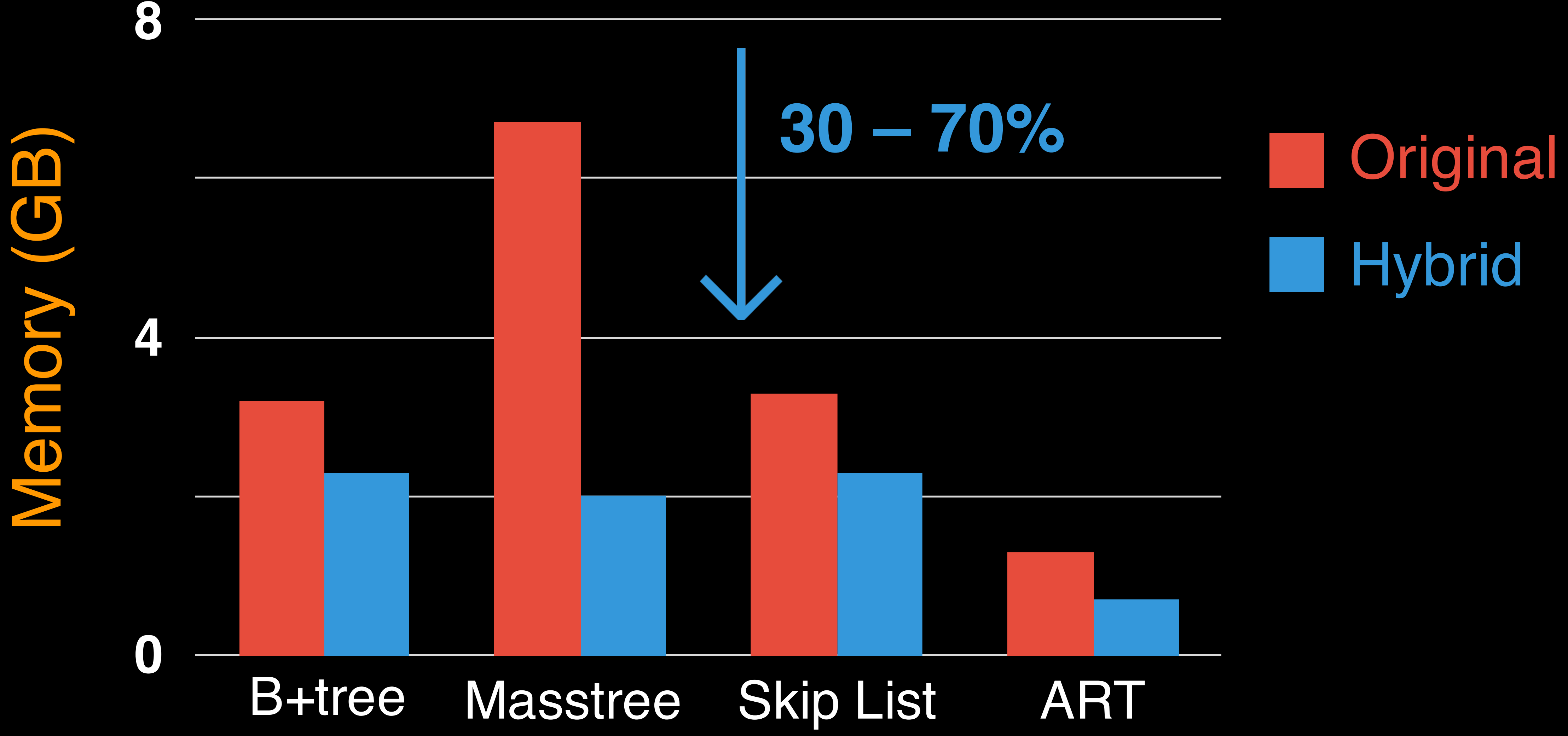
Key: email

Value: 64-bit unsigned integer (pointer)

Single thread

50M entries, **10M** queries (Zipf distributed)

Hybrid index saves memory



Hybrid index provides comparable throughput