Memory-Efficient GroupBy-Aggregate with Compressed Buffer Trees

Hrishikesh Amur

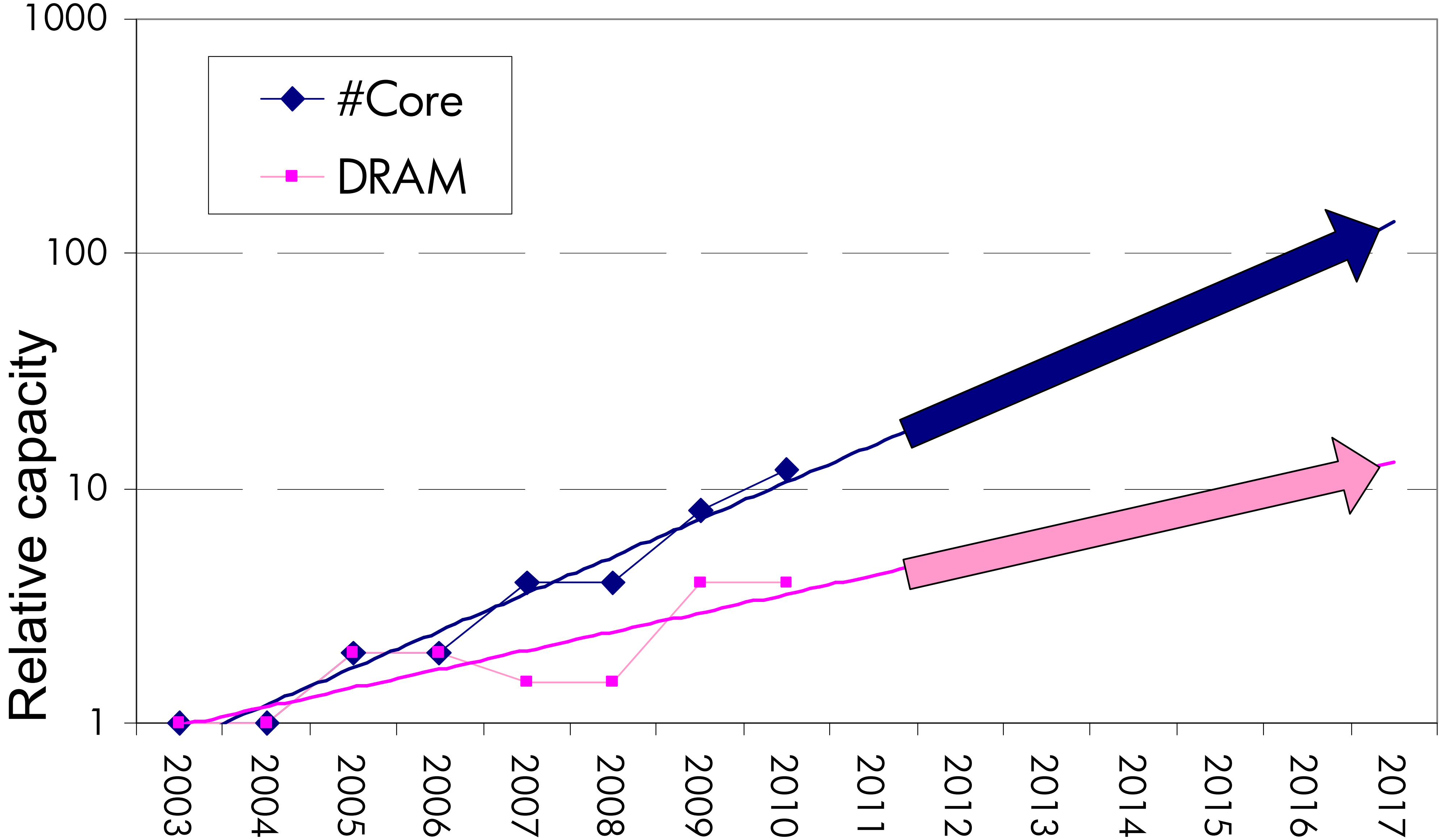
Motivation

Increasing cost of memory

Importance of GroupBy-Aggregate

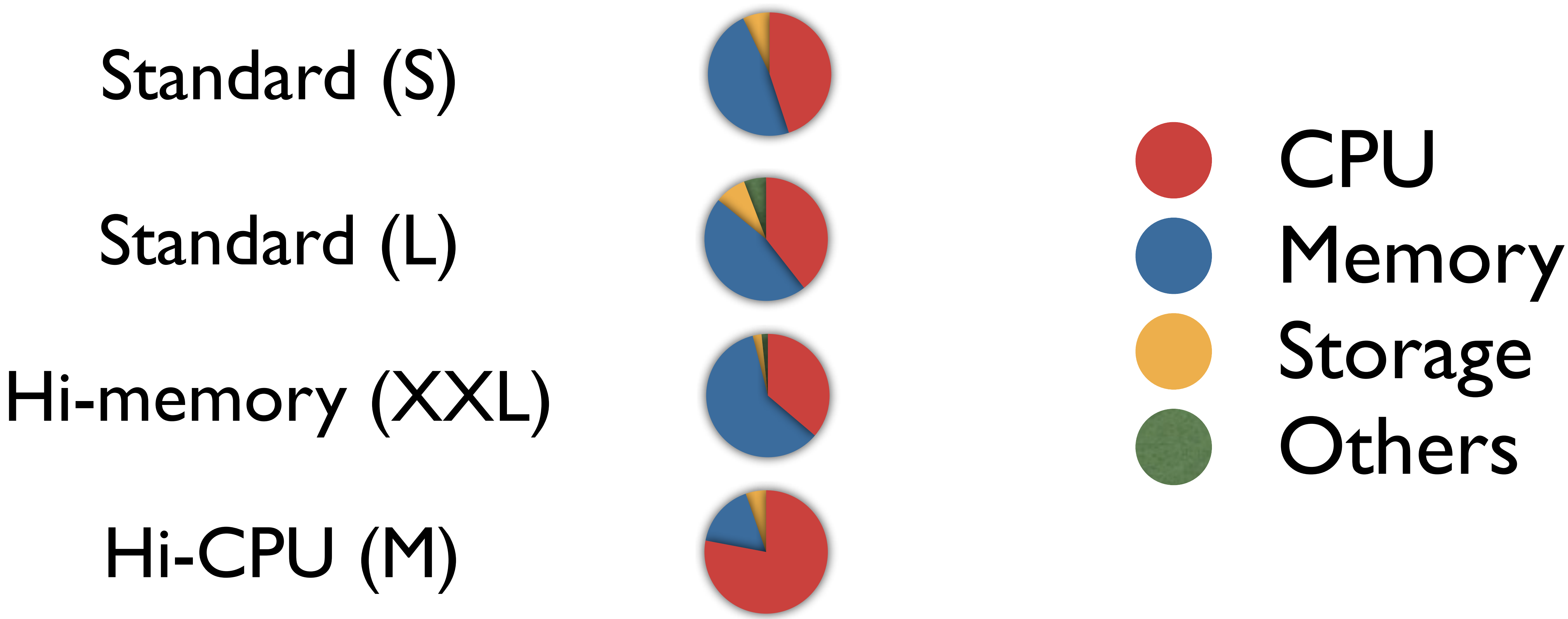
Need for Memory Efficiency

Decreasing memory capacity per core



1. Disaggregated Memory for Expansion and Sharing in Blade Servers, Lim et al., ISCA'09

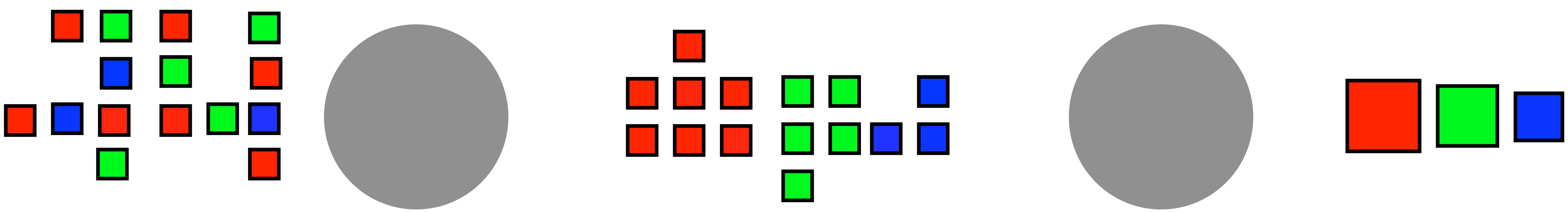
DRAM is expensive



Amazon EC2 proportional resource cost

GroupBy-Aggregate

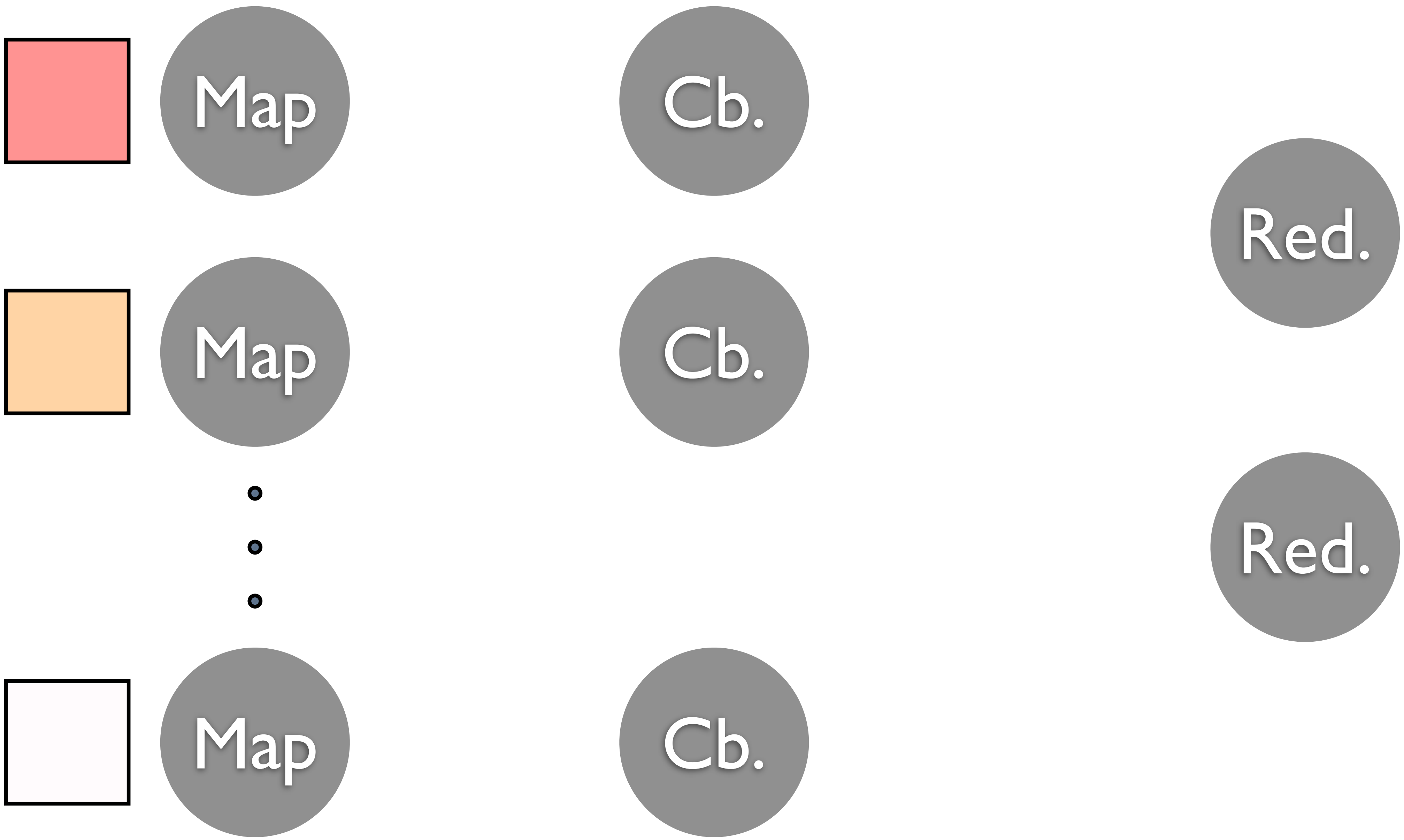
GroupBy-Aggregate



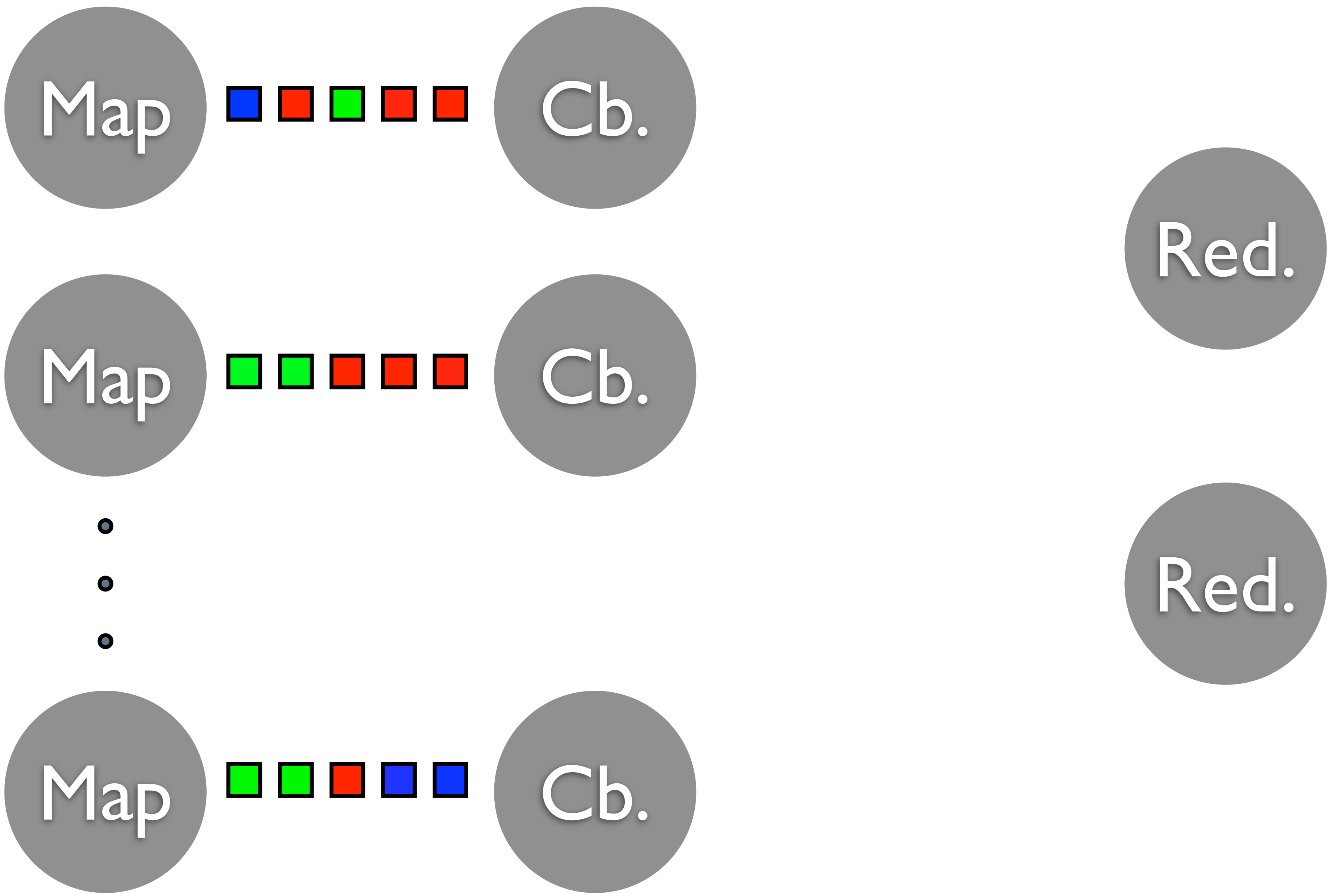
GroupBy Aggregate

- key-value pair

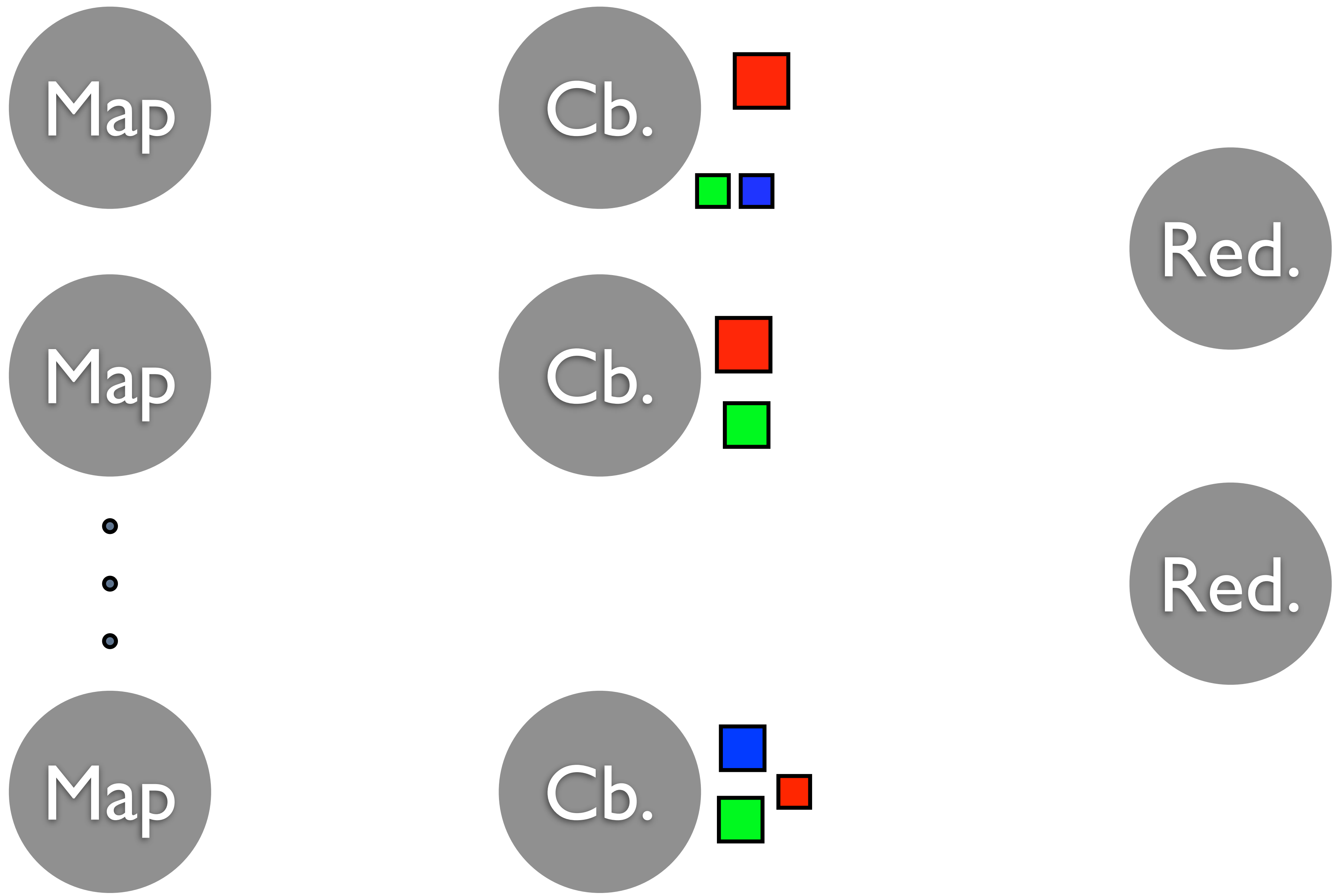
MapReduce



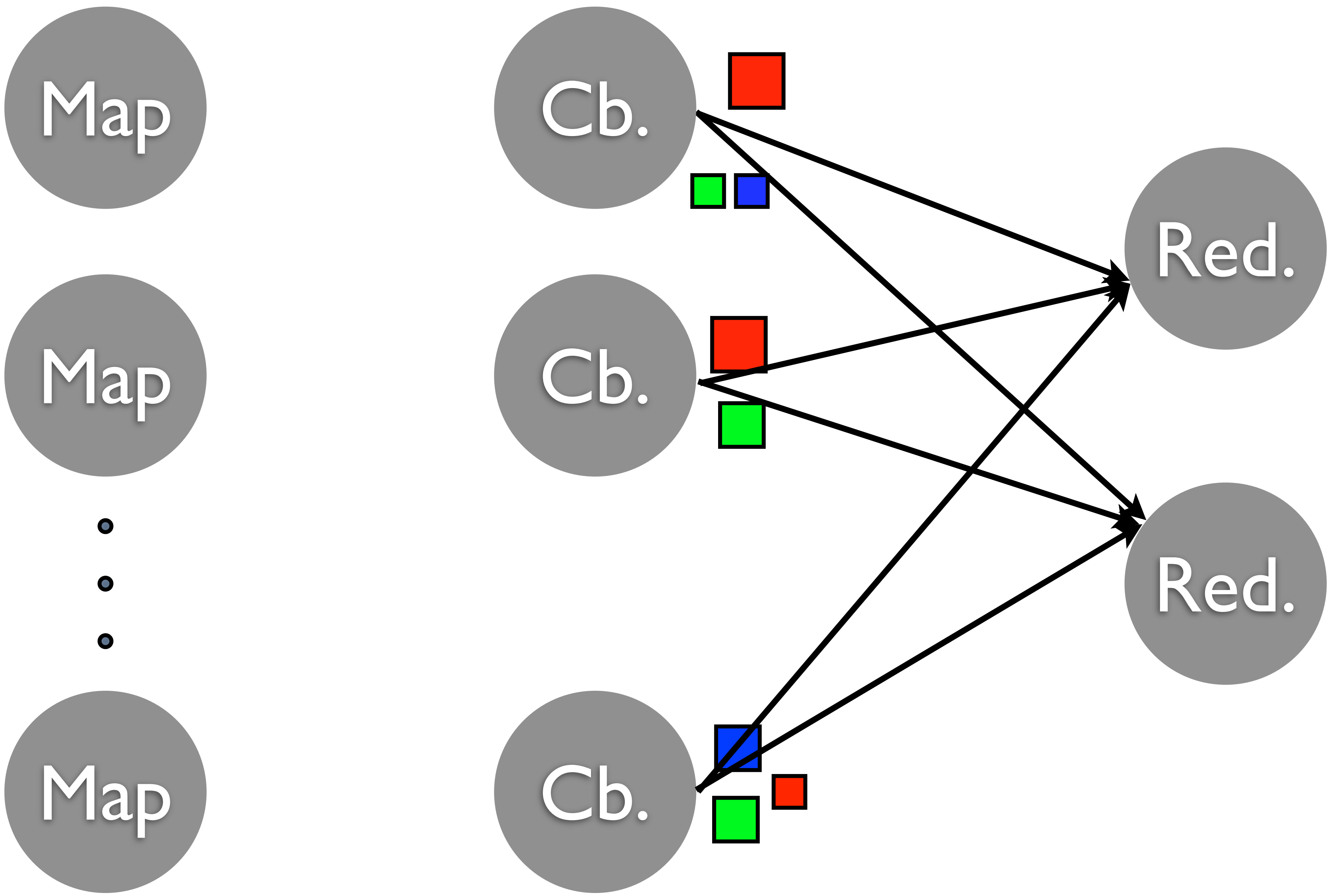
MapReduce



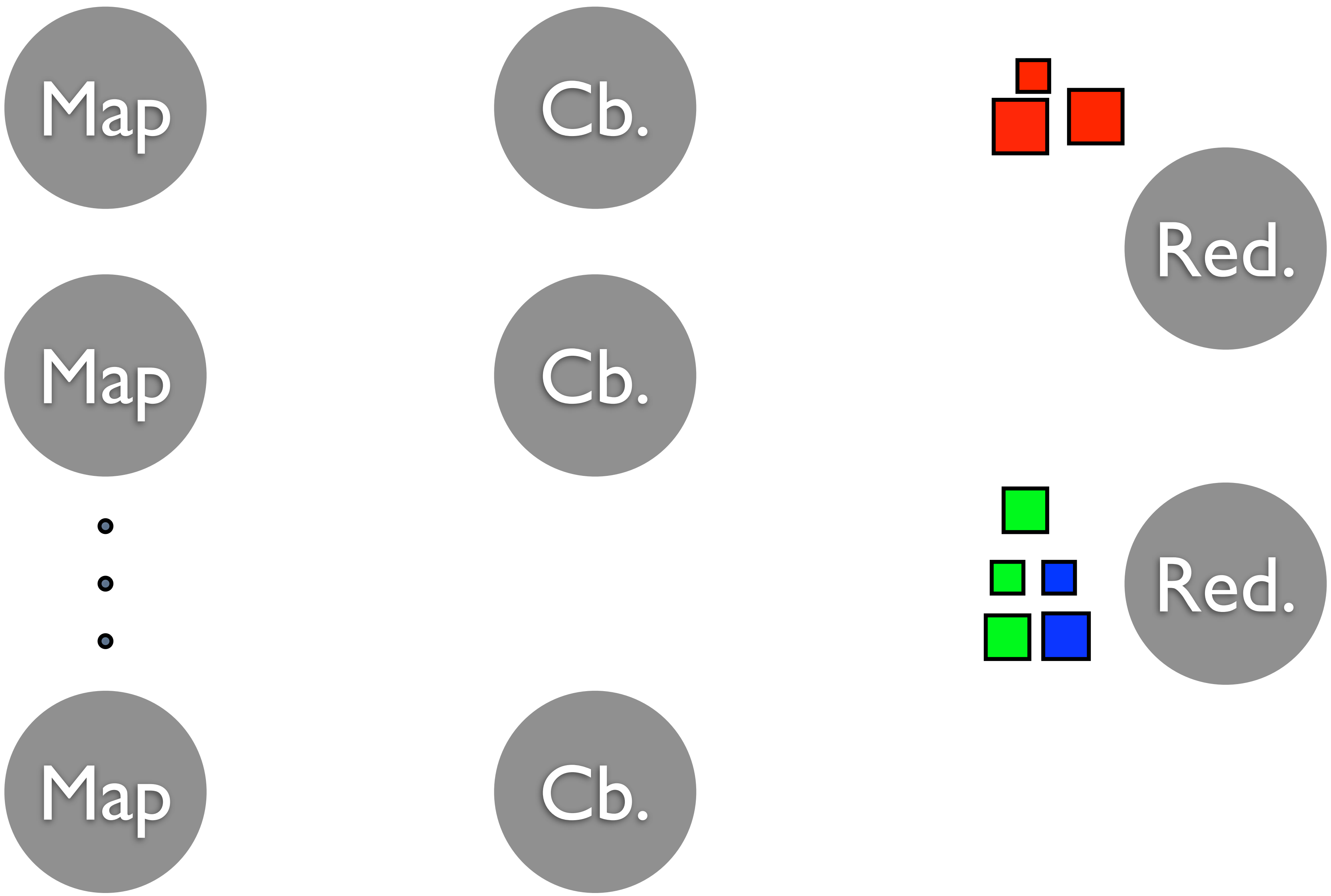
MapReduce



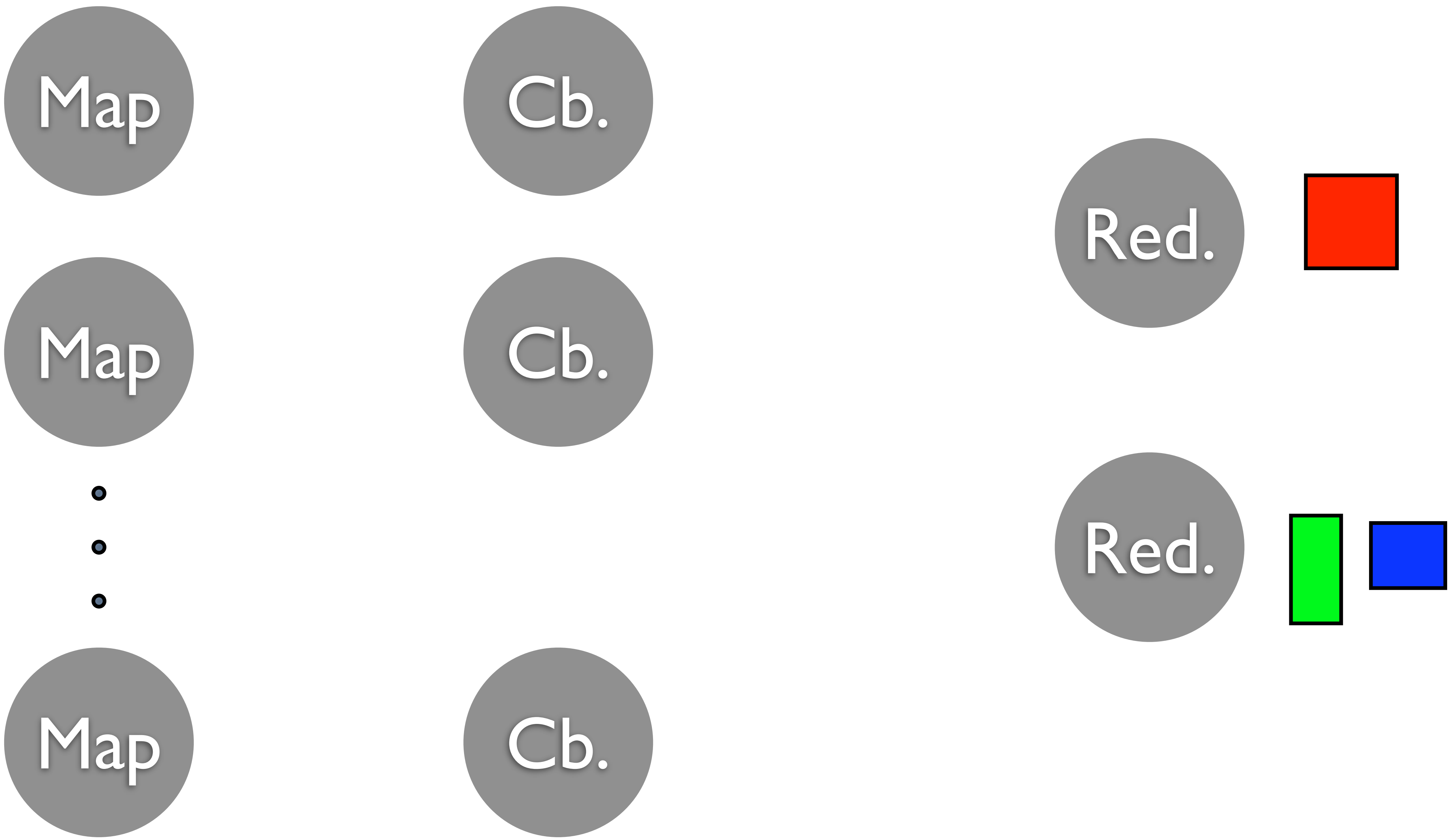
MapReduce



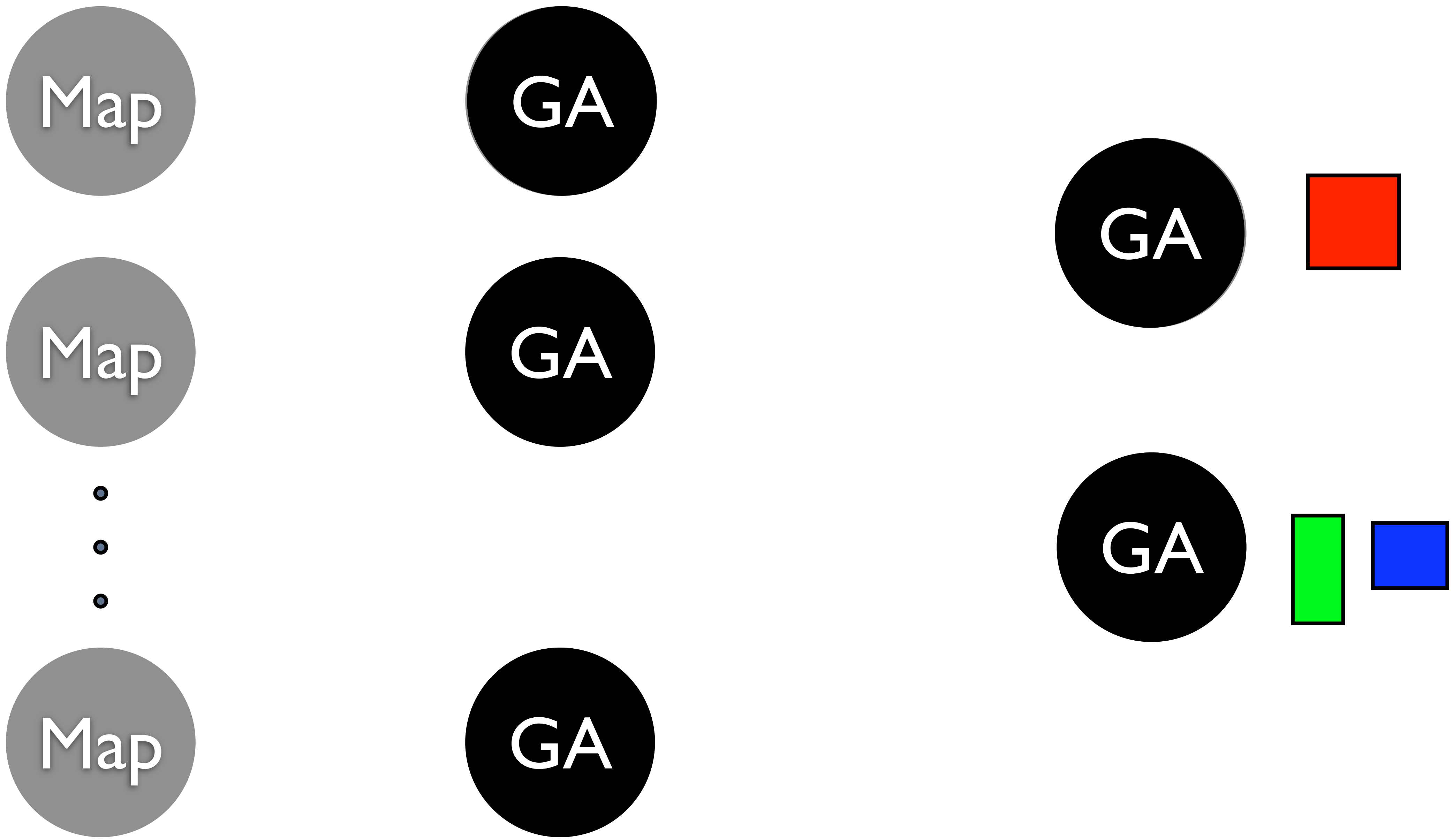
MapReduce



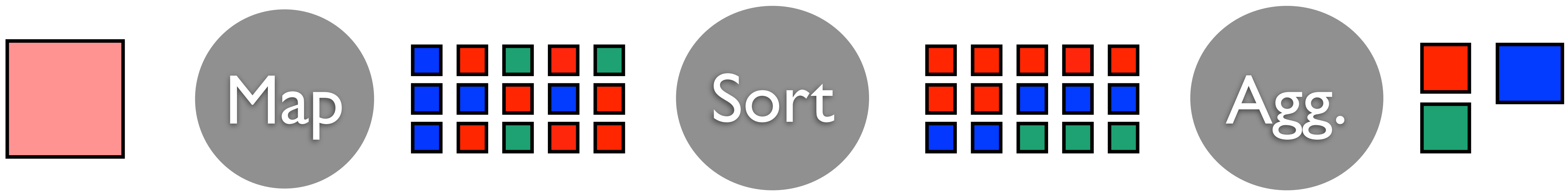
MapReduce



MapReduce



Implementation of G-A



Sort *orders* keys along with *grouping* them

Sort vs. Hash-based GA

Hash typically outperforms Sort for aggregation workloads[1,2,3]

1. Distributed Aggregation for Data-Parallel Computing:Interfaces and Implementations, Yu et. al., SOSP'09
2. Tenzing: A SQL Implementation On The MapReduce Framework, Chattopadhyaya et al., VLDB'11
3. A Platform for Scalable One-Pass Analytics using MapReduce, Li et al., SIGMOD'11

Hash-based G-A requires lots of memory

Hashtable Overheads

Allocator	Per-entry memory (B)	
	std::unordered_map	sparse_hash_map
hoard [9]	64.9	67.8
tcmalloc [21]	57.2	43
jemalloc [20]	58.1	**41**

Dataset:
Key: 8B char array
Value: 4B integer

Sources of Memory Overhead

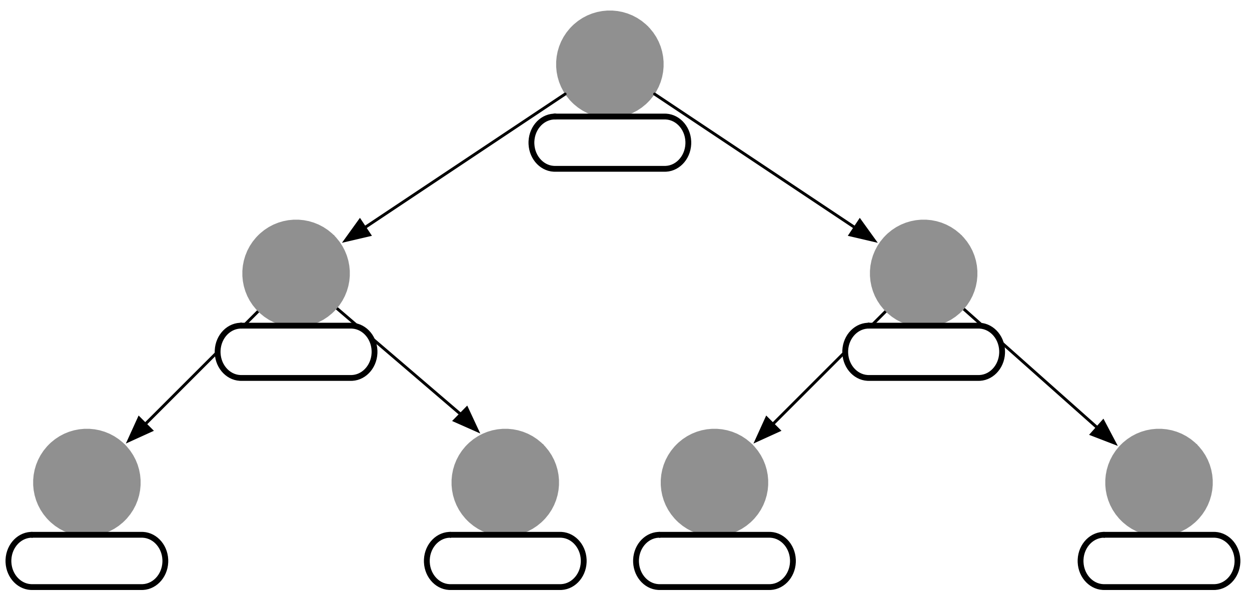
- Allocator overhead for small heap objects
- Indirection overhead (64bit)
- Empty slots in hashtable

How to build a **memory-efficient** and **fast** GroupBy-Aggregate?

Approach

Use Compression for Memory Efficiency

Compressed Buffer Trees (CBT)

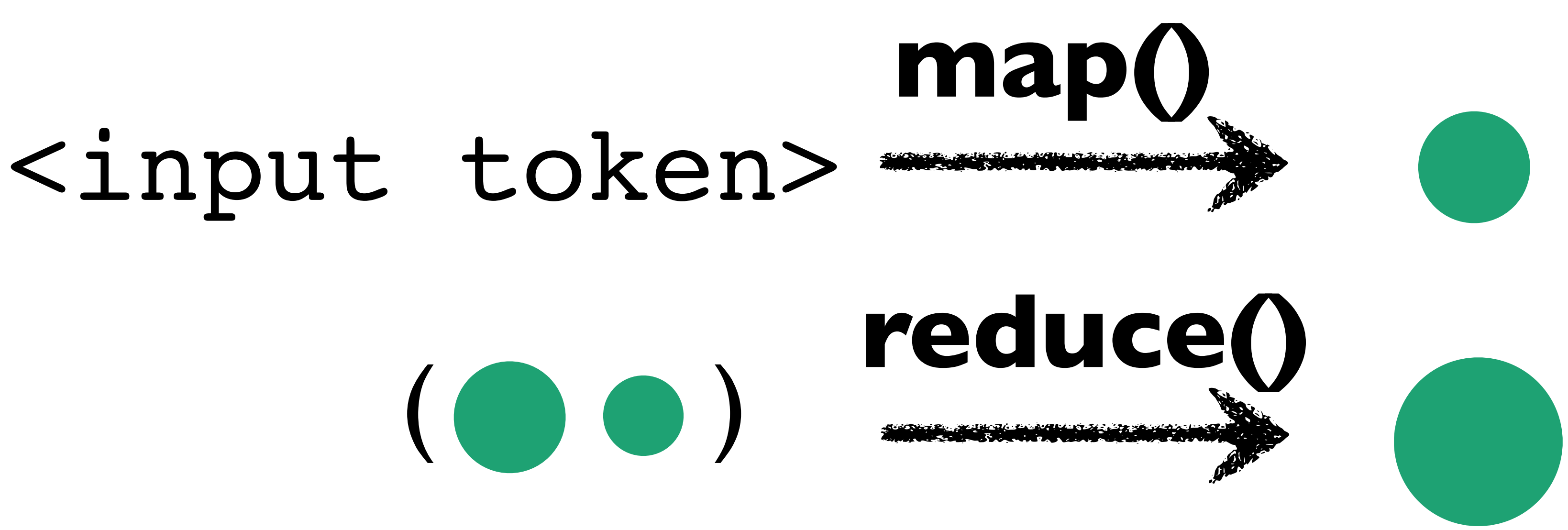


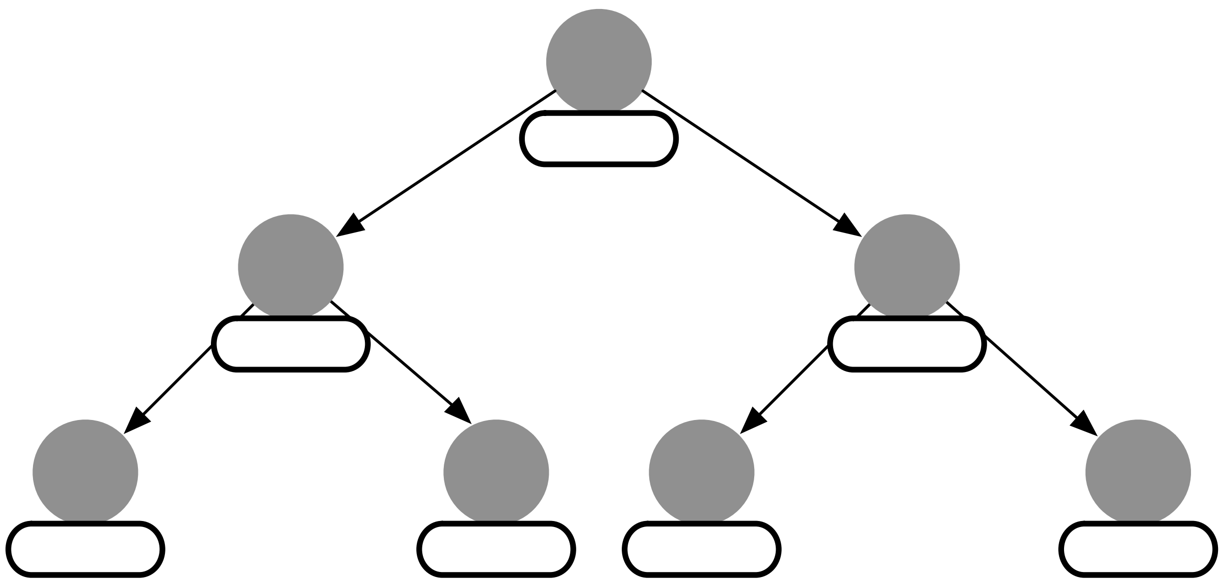
- In-memory B-tree with each node augmented with a memory buffer
- Inspired by the buffer tree[1]

1. The Buffer Tree: A New Technique for Optimal I/O Algorithms, Arge.

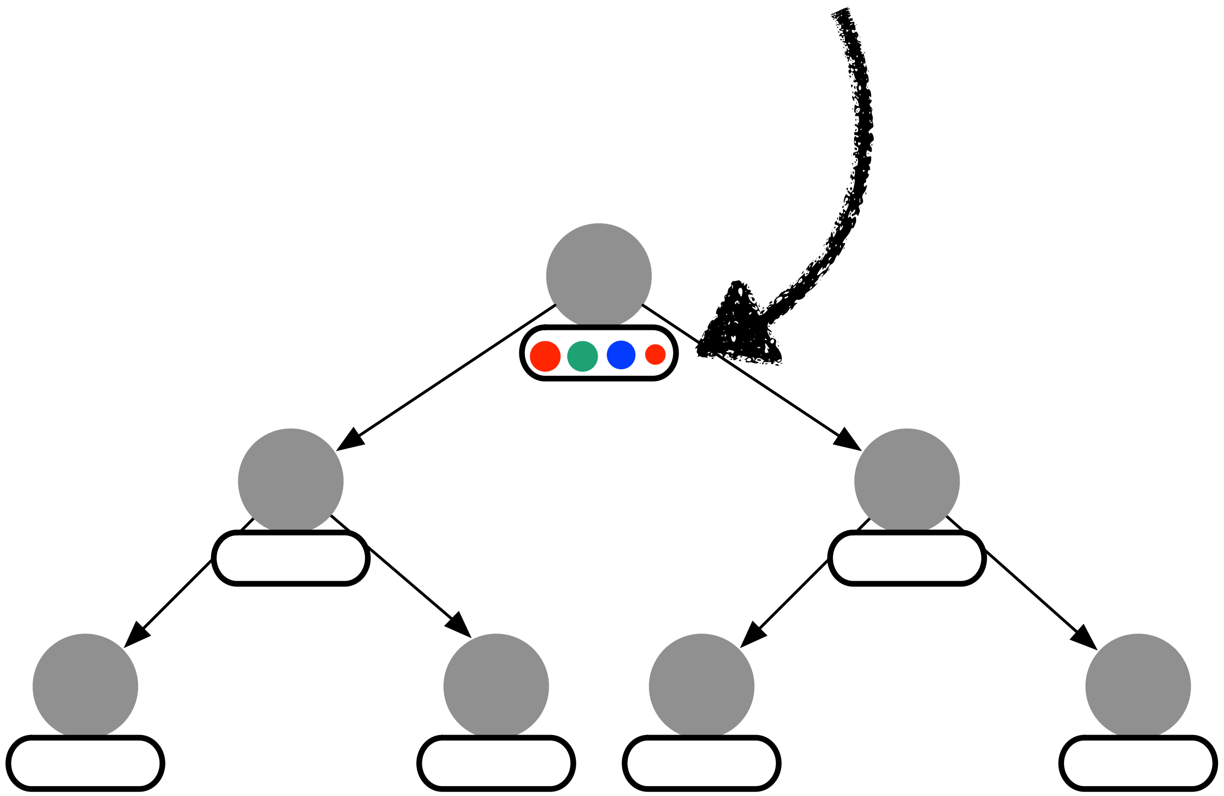
Terminology

- **Partial Aggregation Object (PAO)**
  - User-defined key and value
  - Eg. (char*, uint32) for wordcount, (char*, vector<T>) for k-Nearest-Neighbor

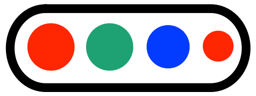


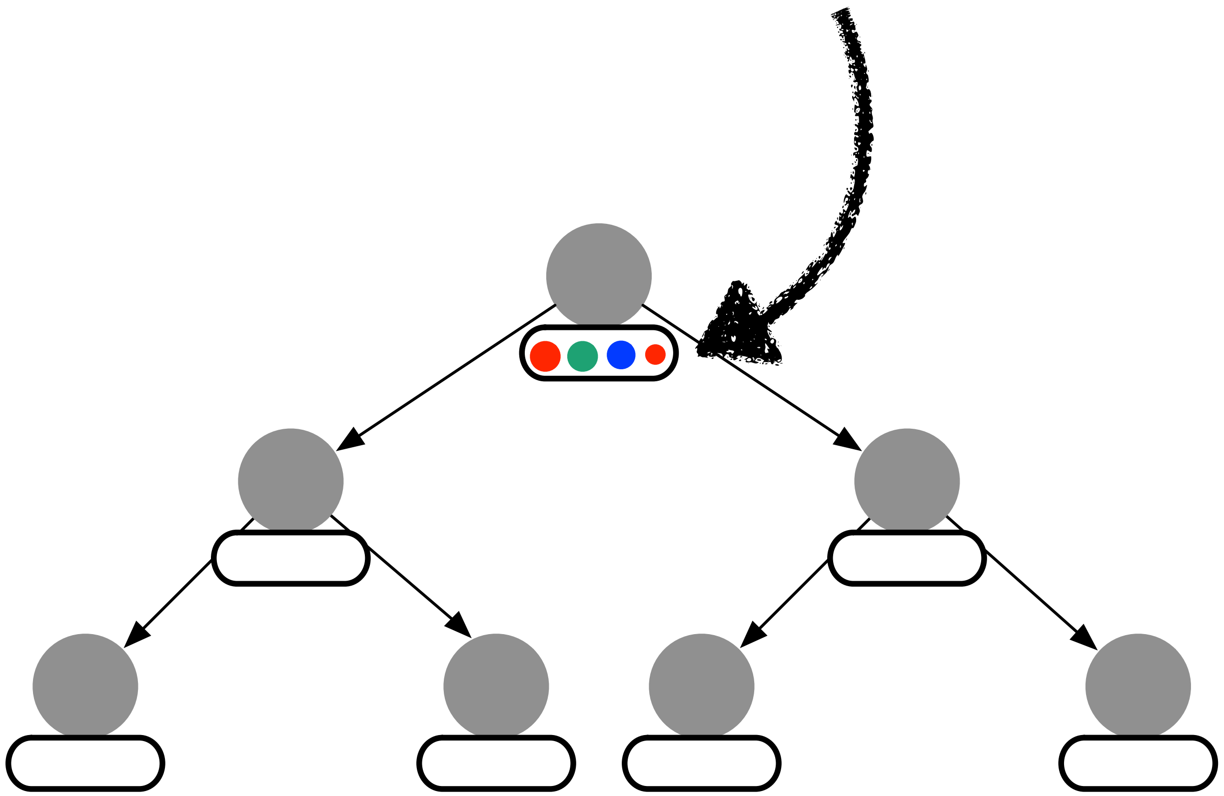


① Insert **PAO**



① Insert **PAO** ② Full root:

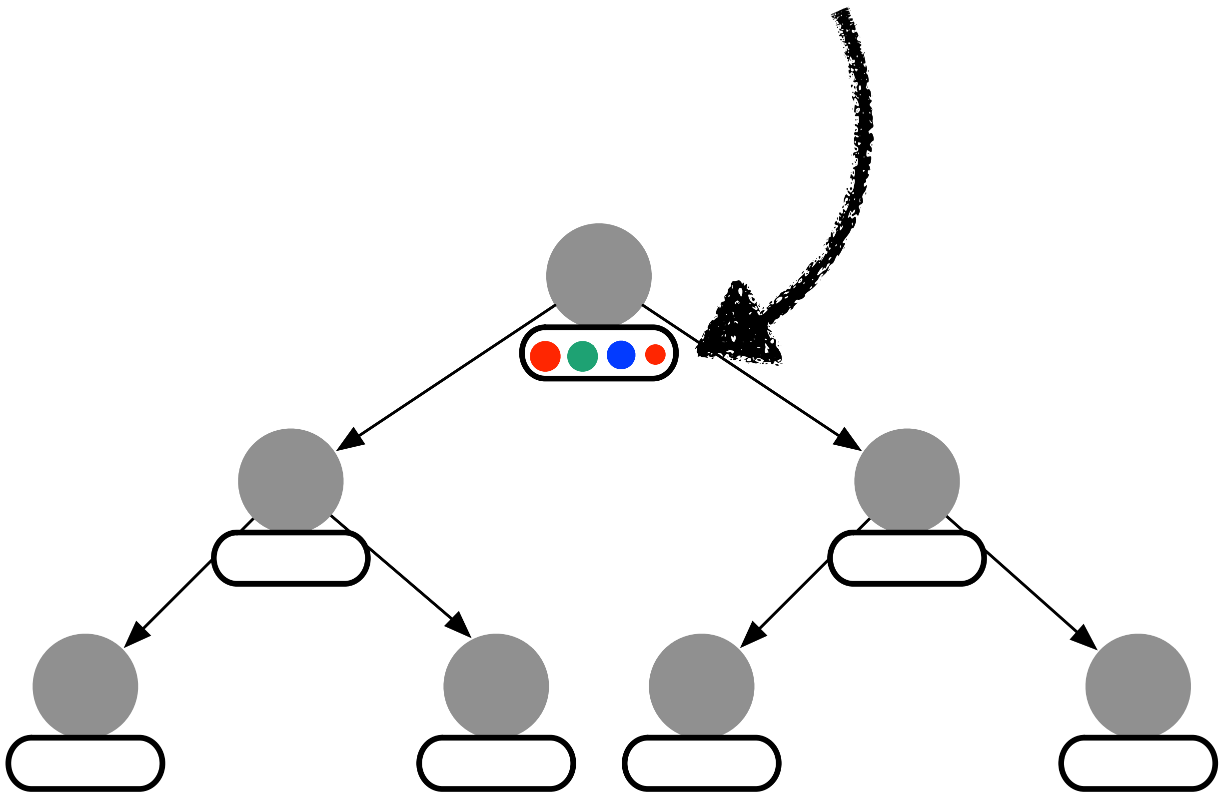




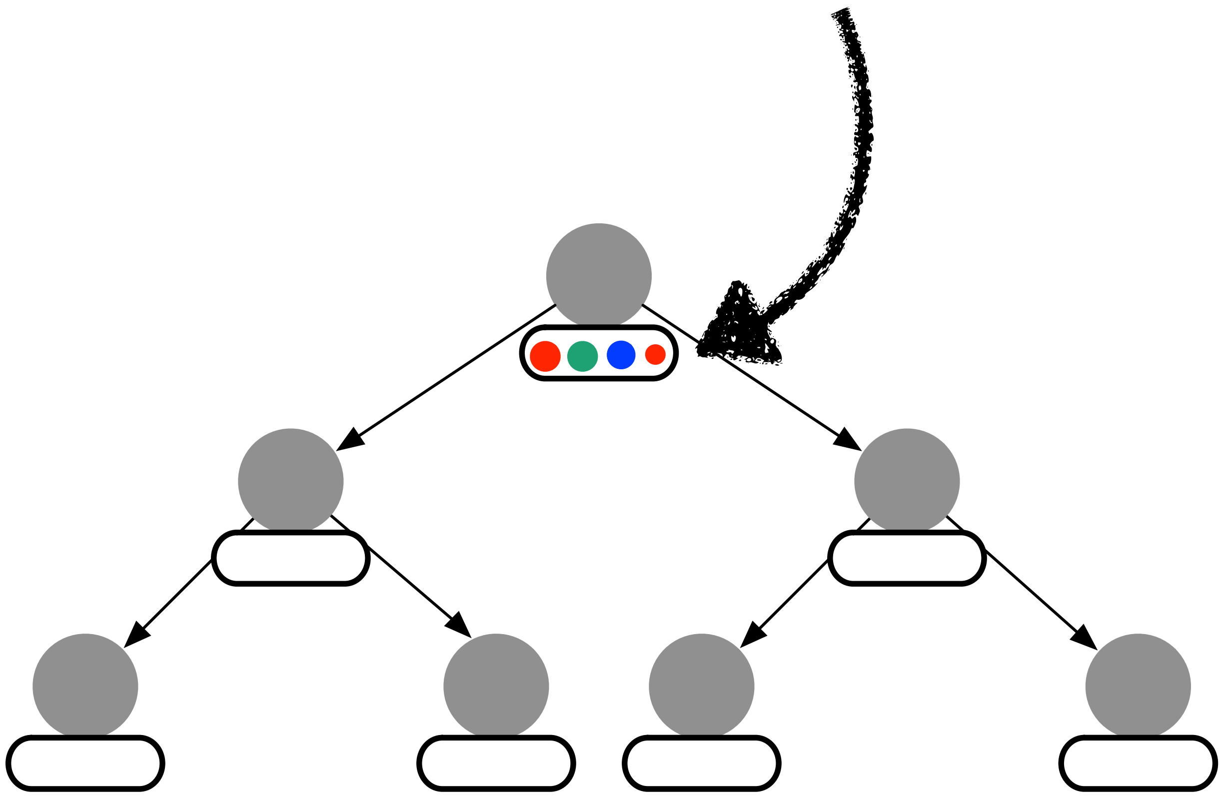
① Insert **PAO**

② Full root:

a. sorted



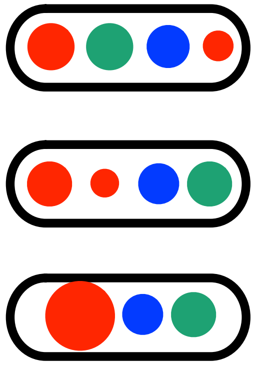
① Insert **PAO**



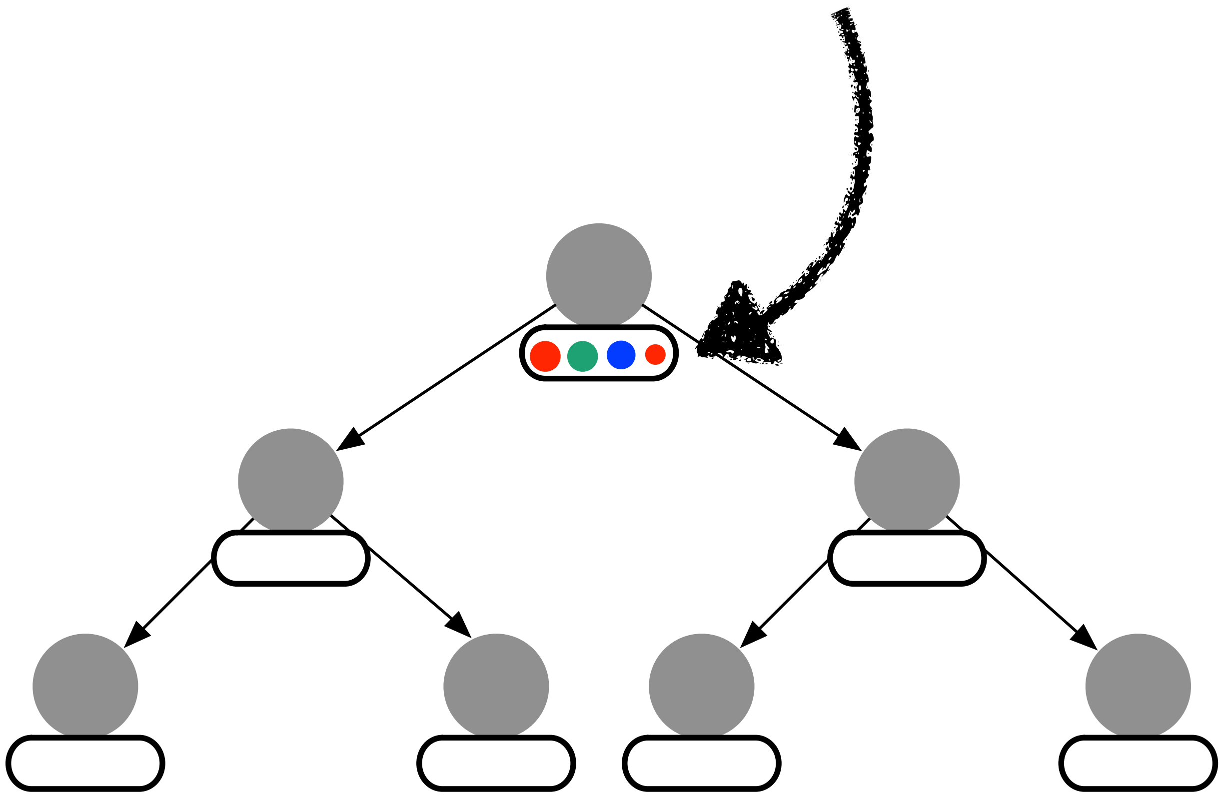
② Full root:

a. sorted

b. aggregated



① Insert **PAO**

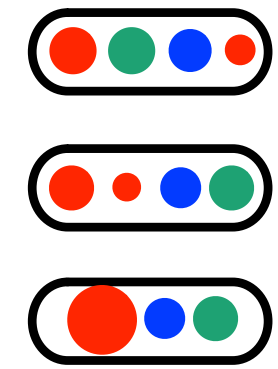


② Full root:

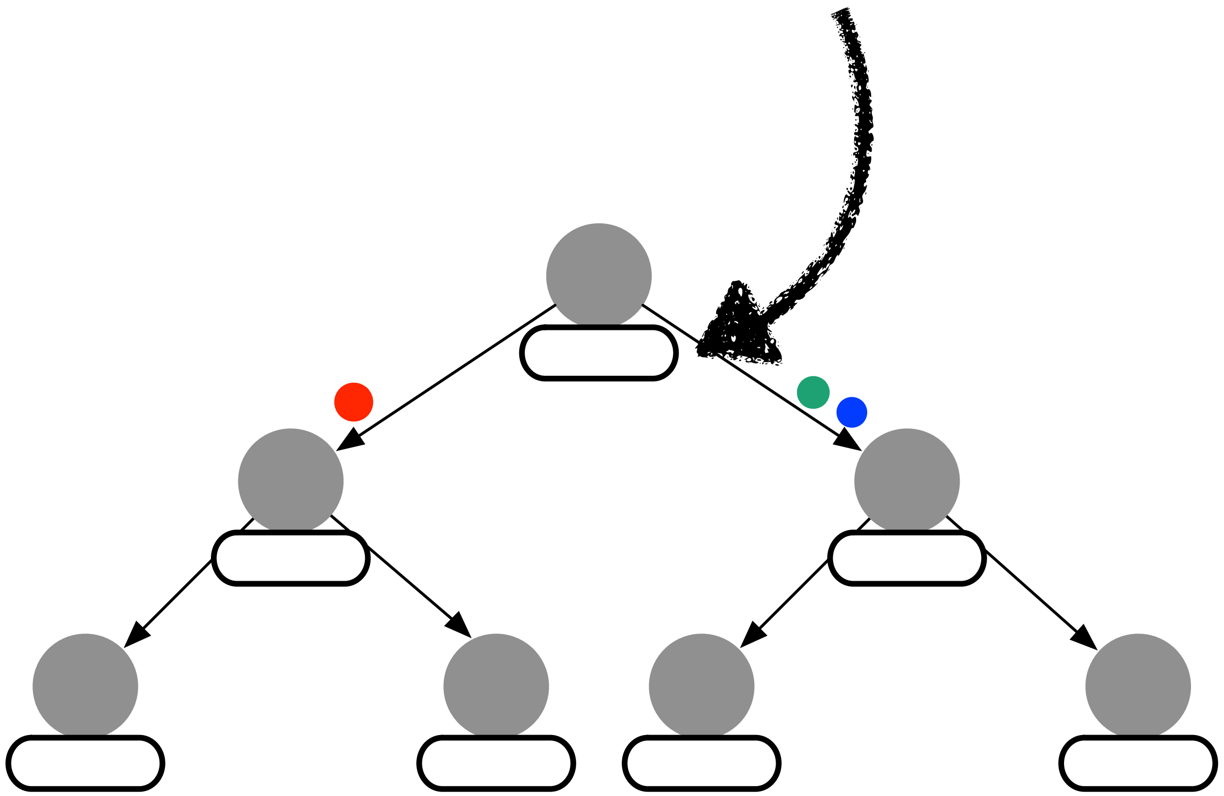
a. sorted

b. aggregated

c. spilled



① Insert **PAO**

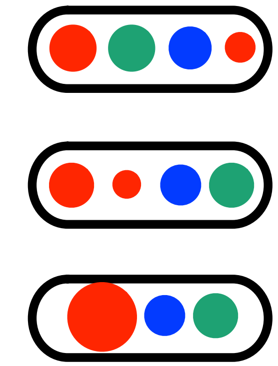


② Full root:

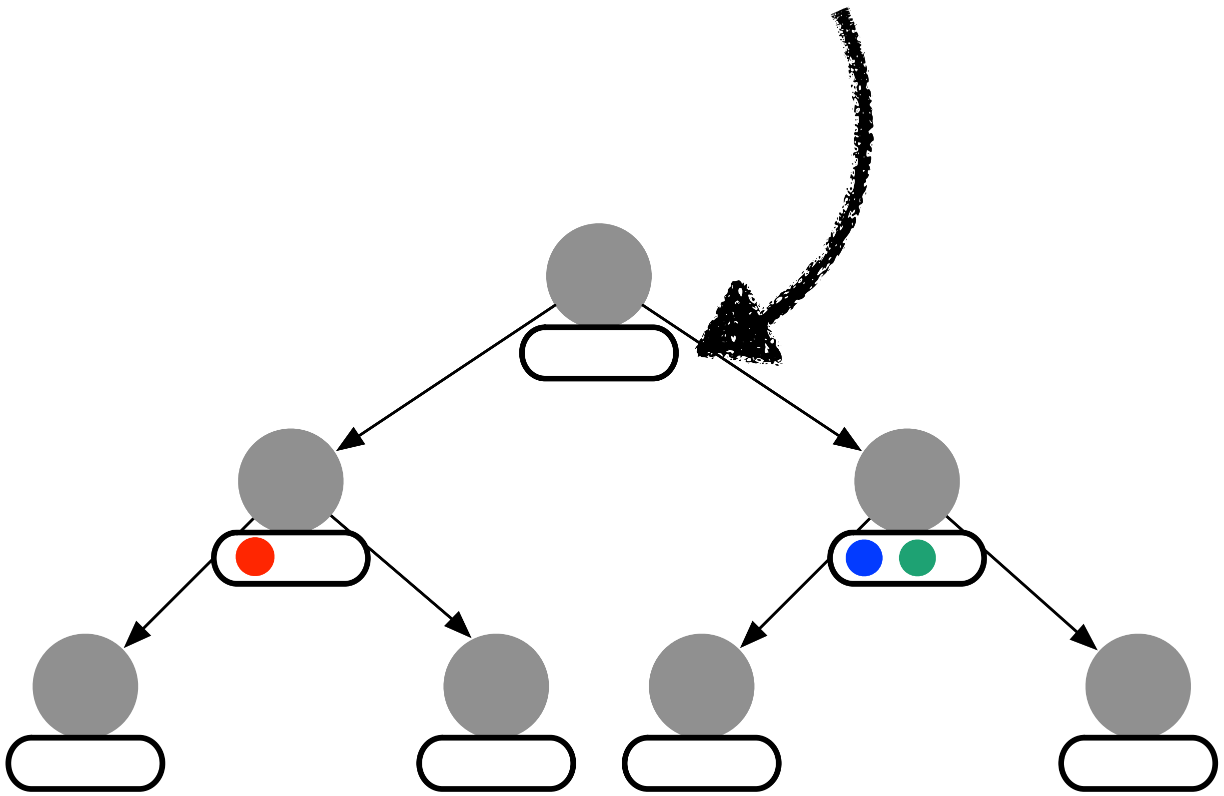
a. sorted

b. aggregated

c. spilled



① Insert **PAO**



② Full root:

a. sorted

b. aggregated

c. spilled

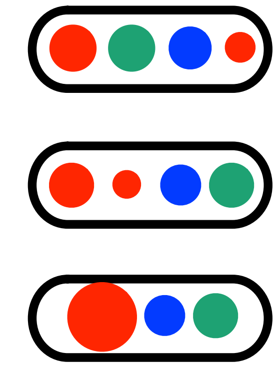
① Insert **PAO**

② Full root:

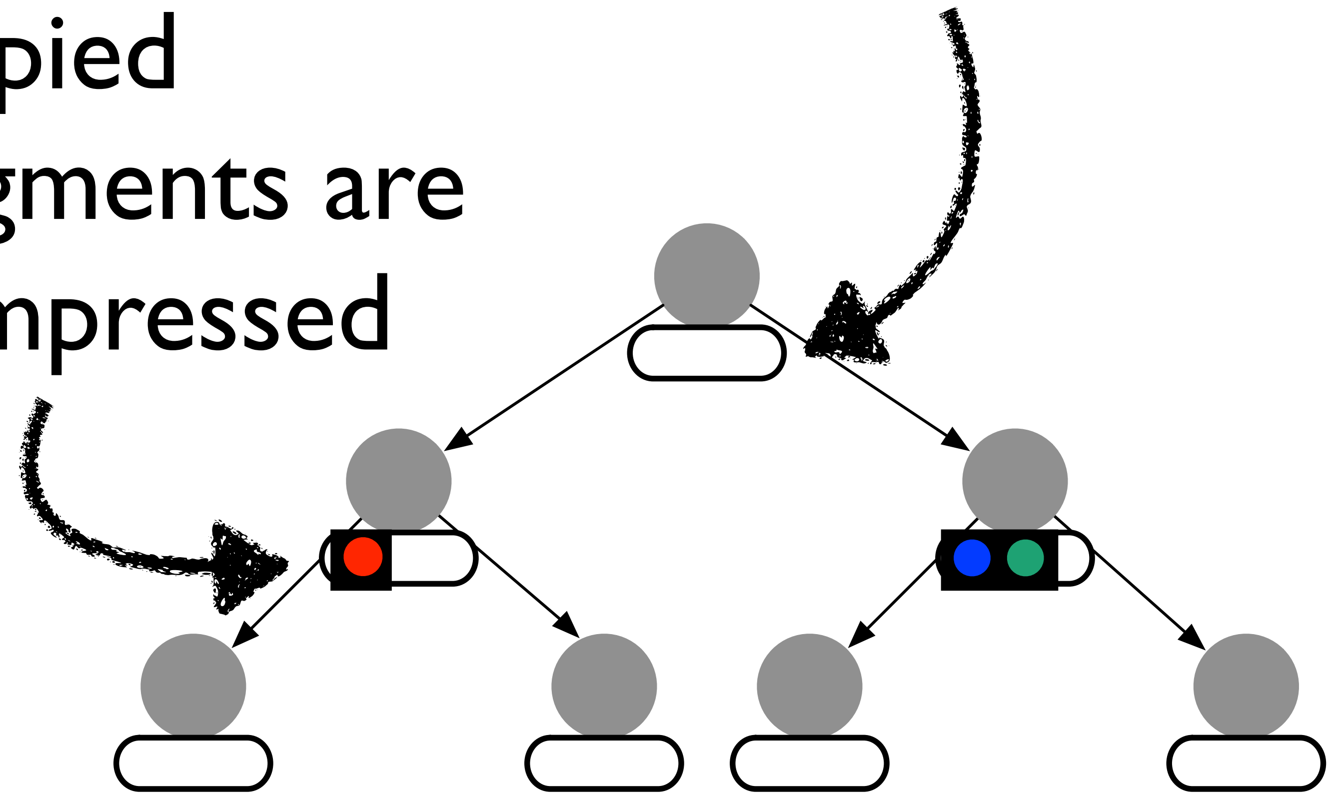
a. sorted

b. aggregated

c. spilled



③ Copied fragments are compressed



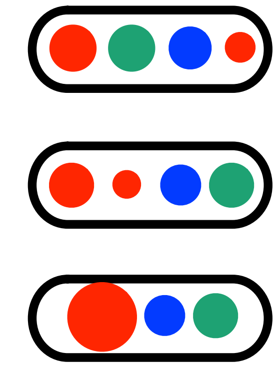
① Insert **PAO**

② Full root:

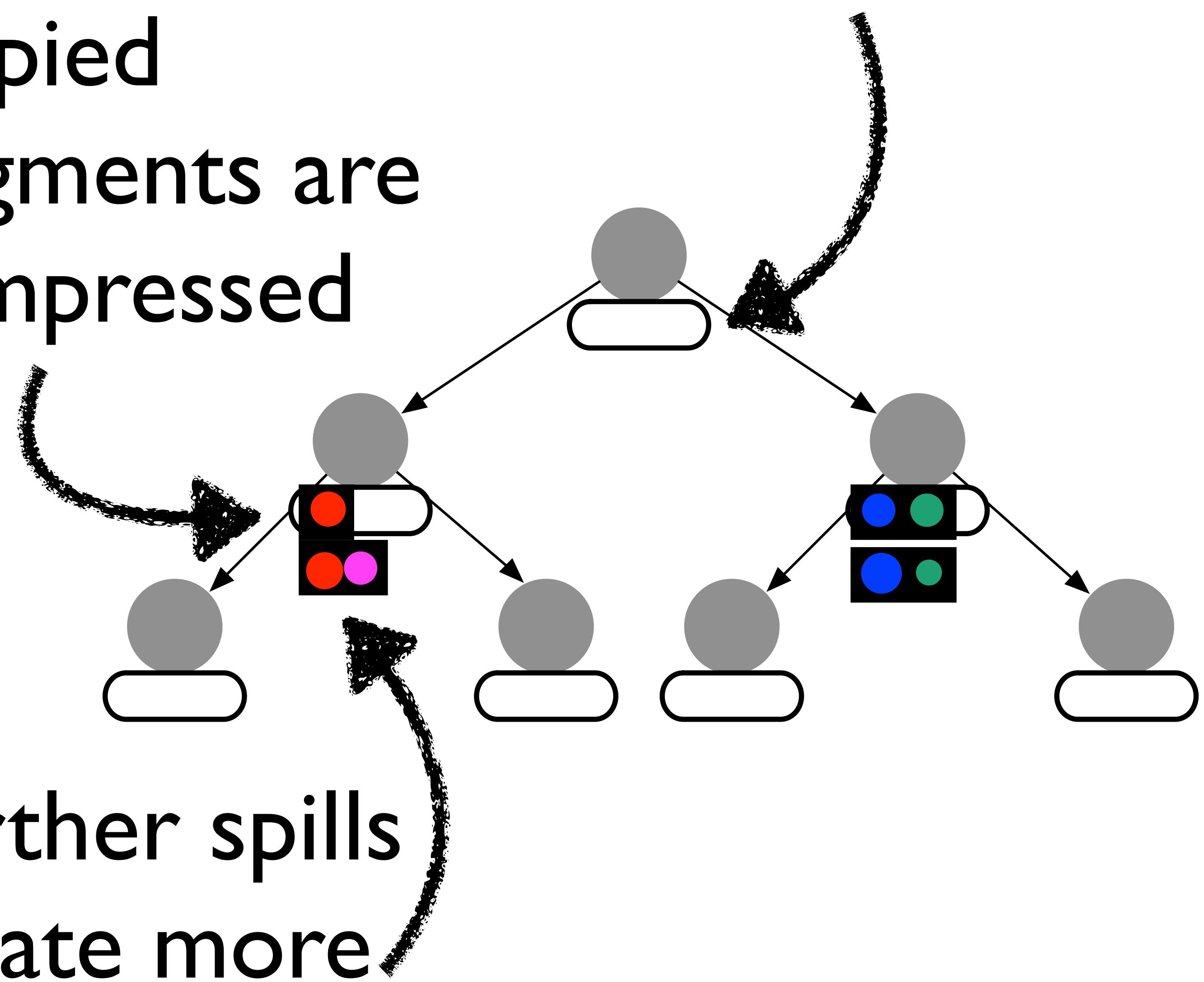
a. sorted

b. aggregated

c. spilled



③ Copied fragments are compressed



④ Further spills create more fragments

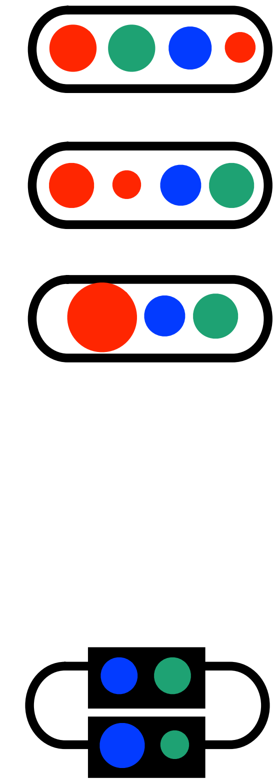
① Insert **PAO**

② Full root:

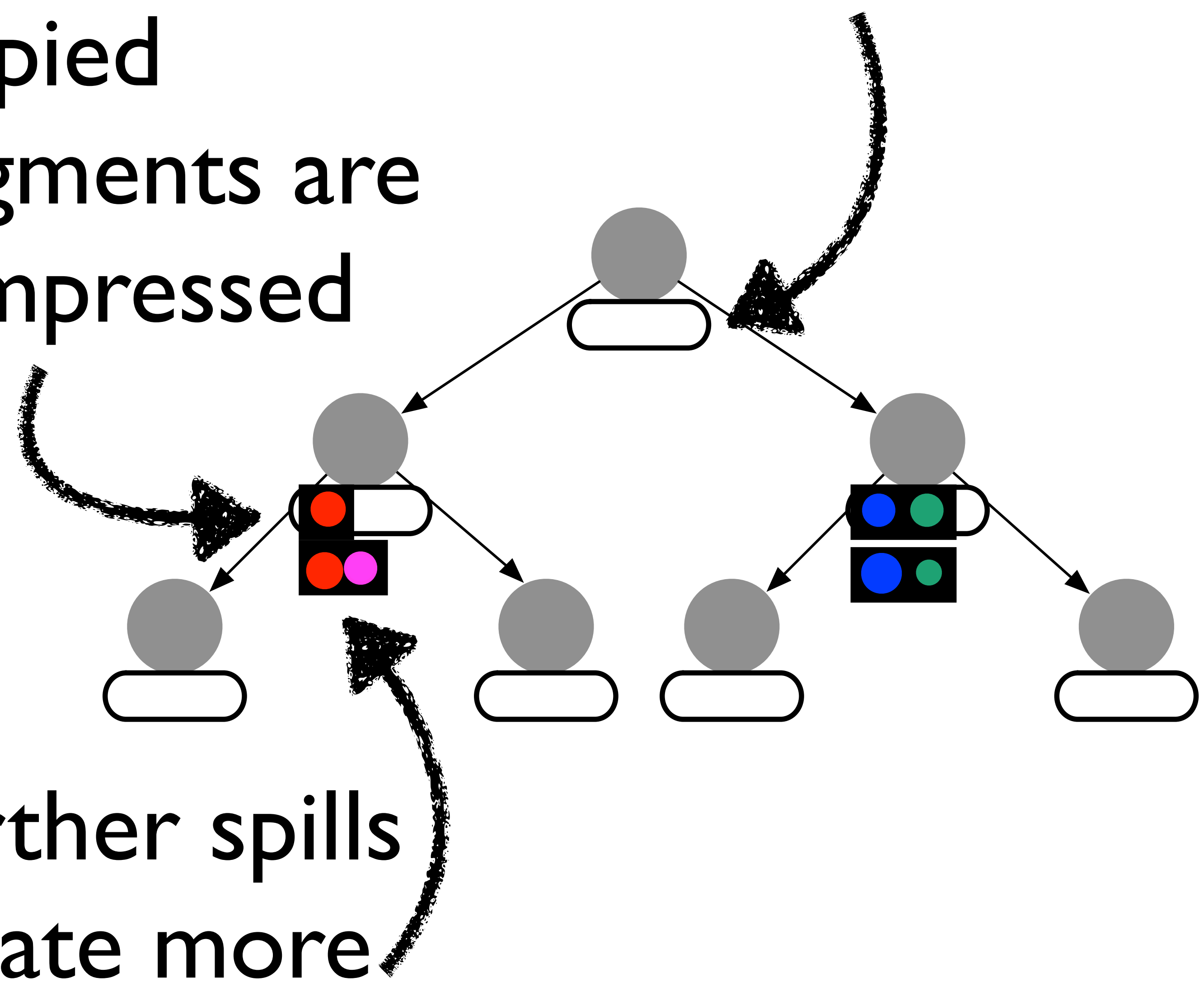
a. sorted

b. aggregated

c. spilled

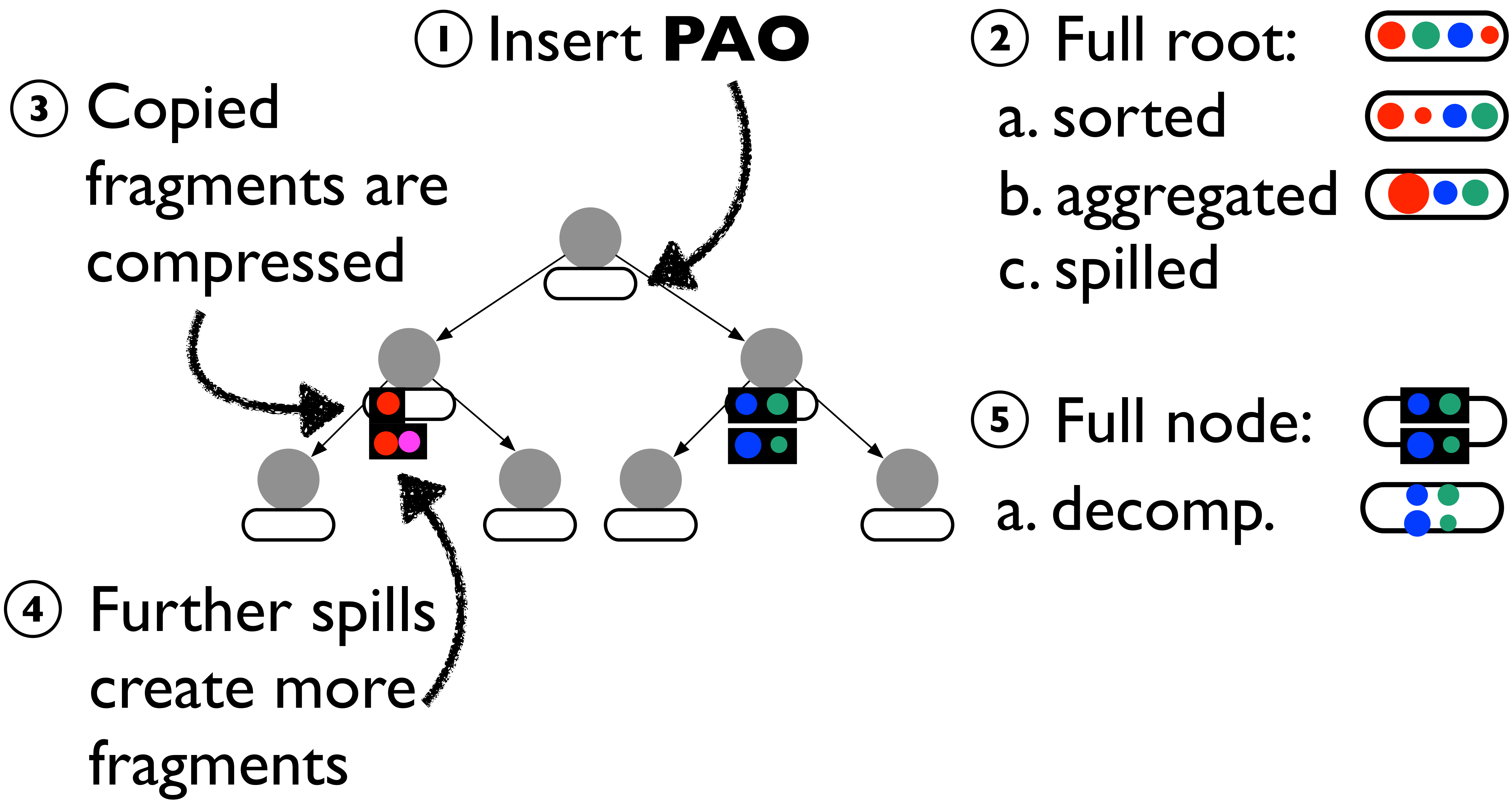


③ Copied fragments are compressed



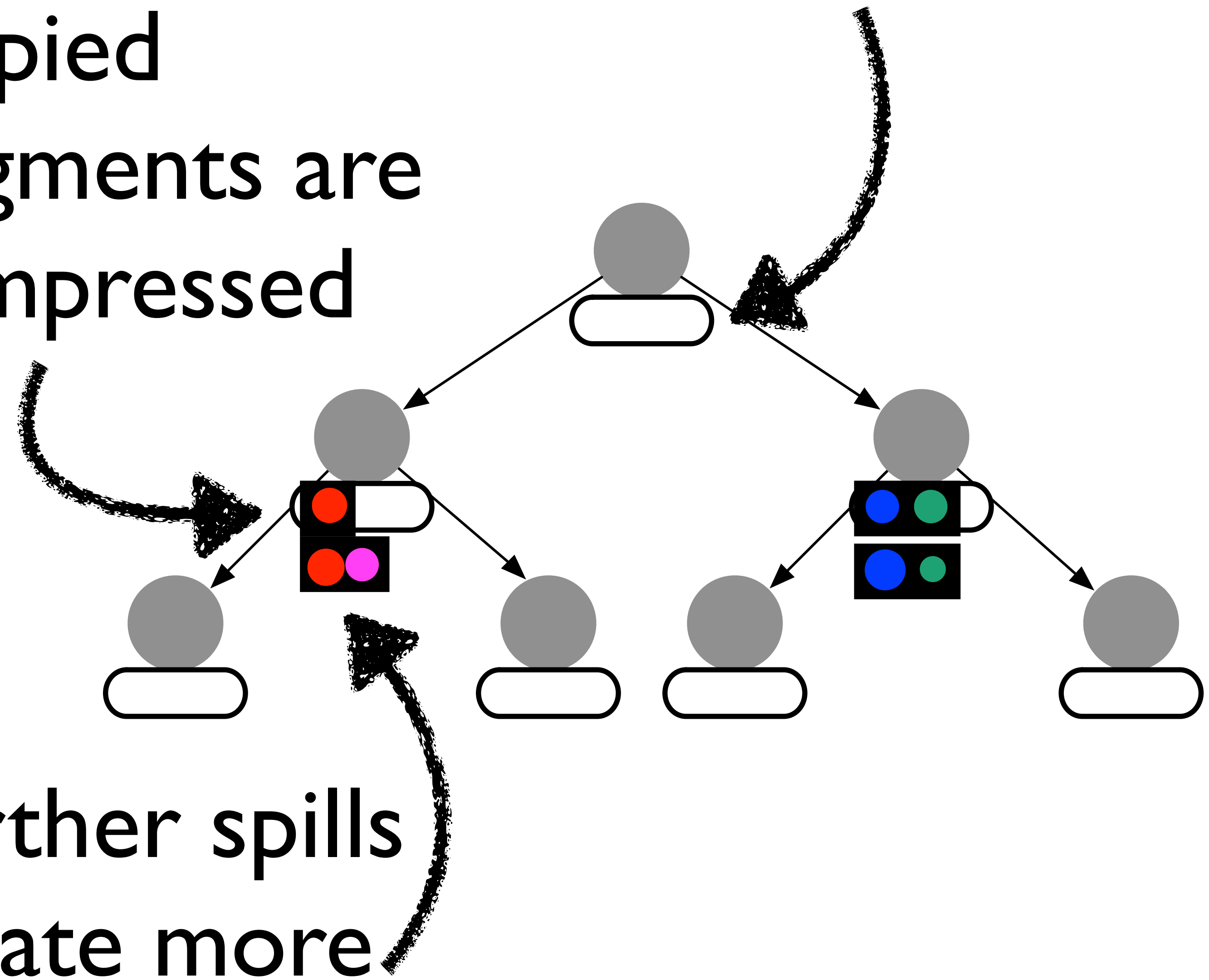
④ Further spills create more fragments

⑤ Full node:


① Insert **PAO**
② Full root:
a. sorted
b. aggregated
c. spilled
③ Copied fragments are compressed
④ Further spills create more fragments
⑤ Full node:
a. decomp.

① Insert **PAO**

③ Copied fragments are compressed



④ Further spills create more fragments

② Full root:

a. sorted

b. aggregated

c. spilled

⑤ Full node:

a. decomp.

b. merged

① Insert **PAO**

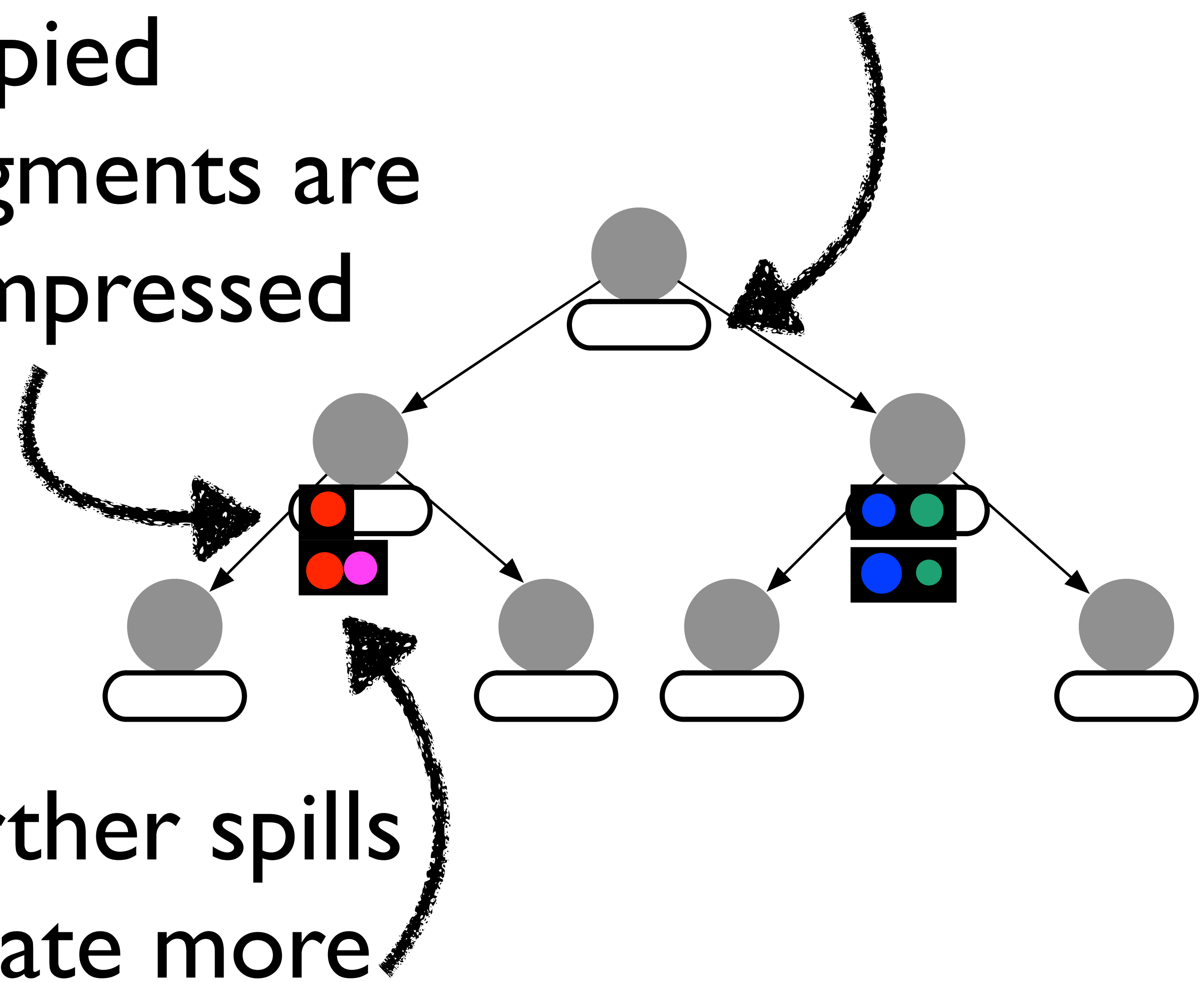
② Full root:

a. sorted

b. aggregated

c. spilled

③ Copied fragments are compressed



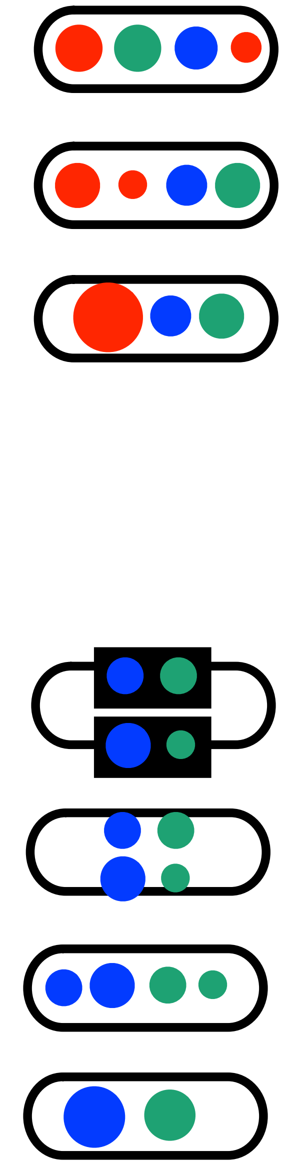
④ Further spills create more fragments

⑤ Full node:

a. decomp.

b. merged

c. aggregated



① Insert **PAO**

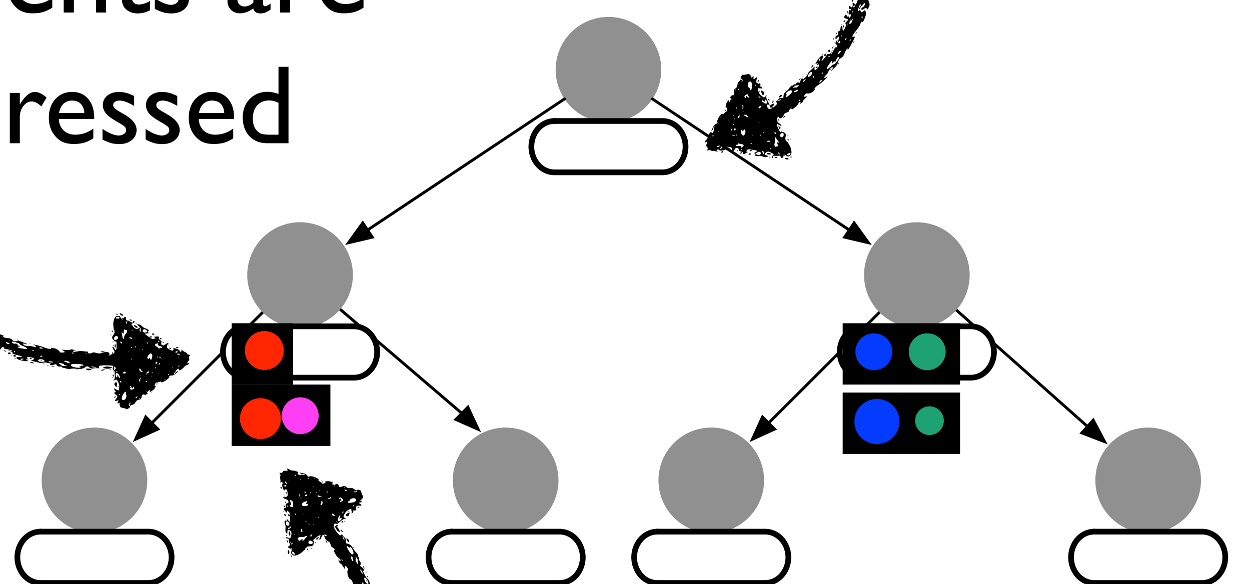
② Full root:

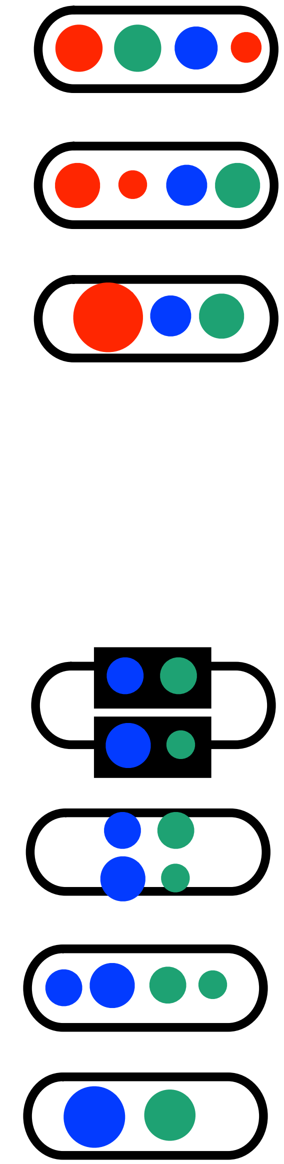
a. sorted

b. aggregated

c. spilled

③ Copied fragments are compressed





④ Further spills create more fragments

⑤ Full node:

a. decomp.

b. merged

c. aggregated

d. spilled

① Insert **PAO**

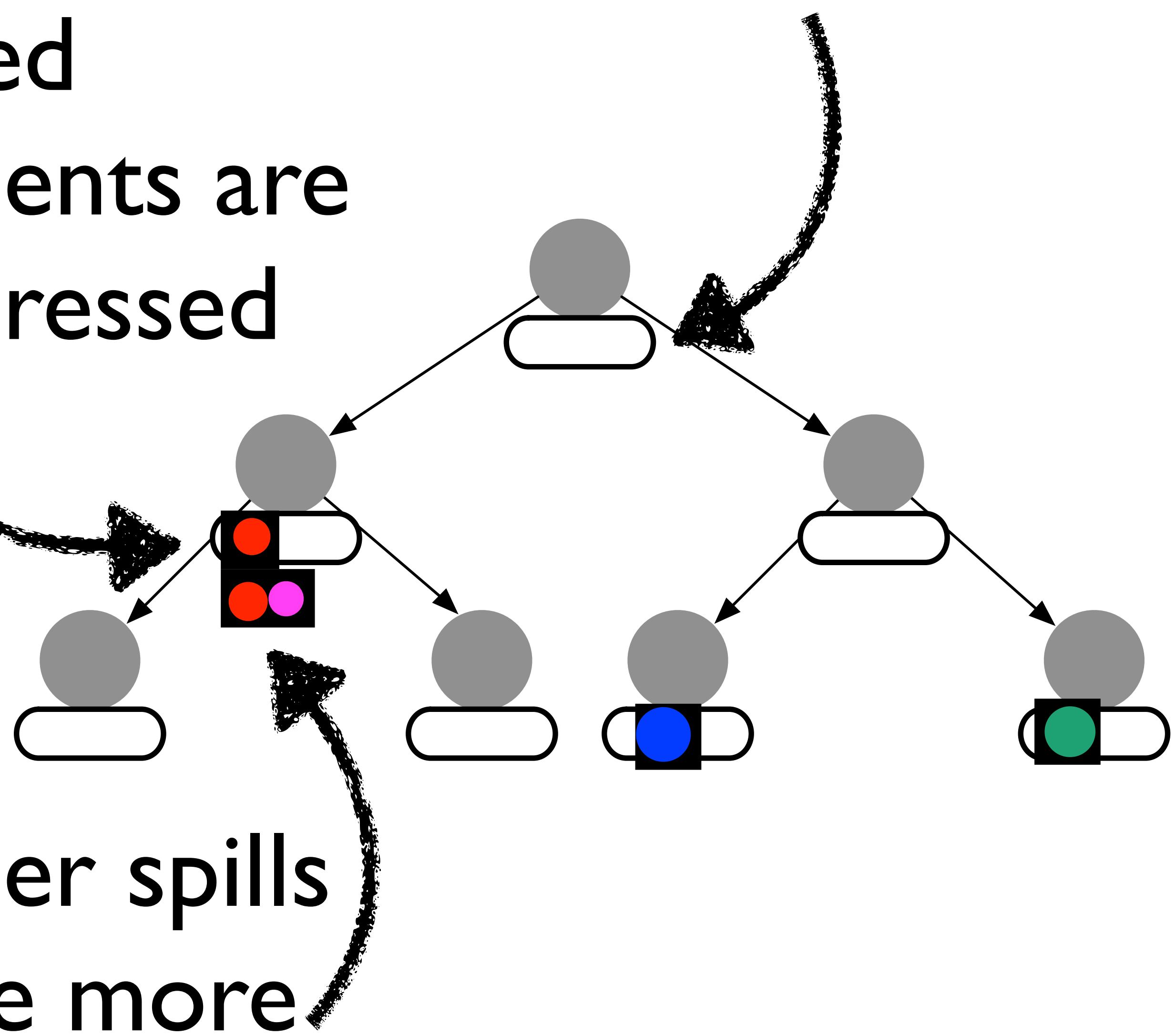
② Full root:

a. sorted

b. aggregated

c. spilled

③ Copied fragments are compressed



④ Further spills create more fragments

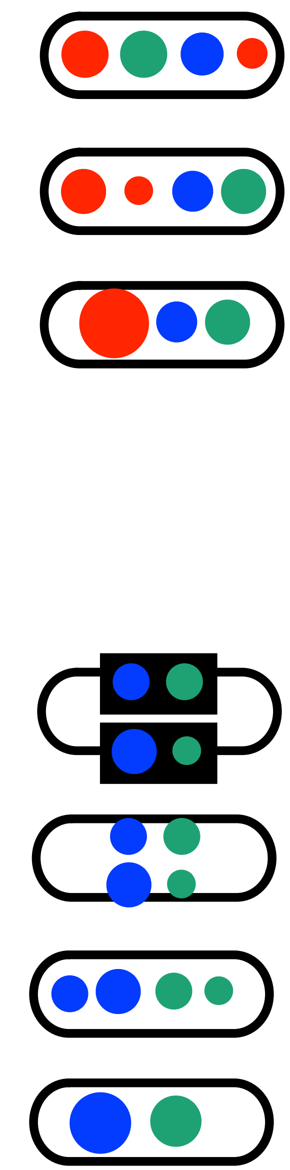
⑤ Full node:

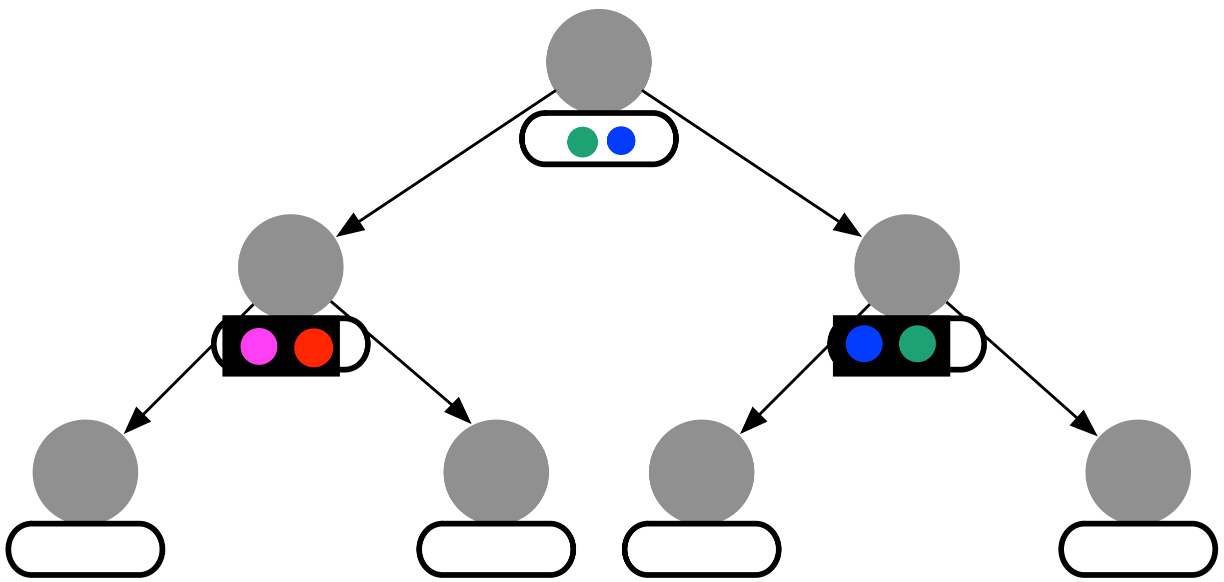
a. decomp.

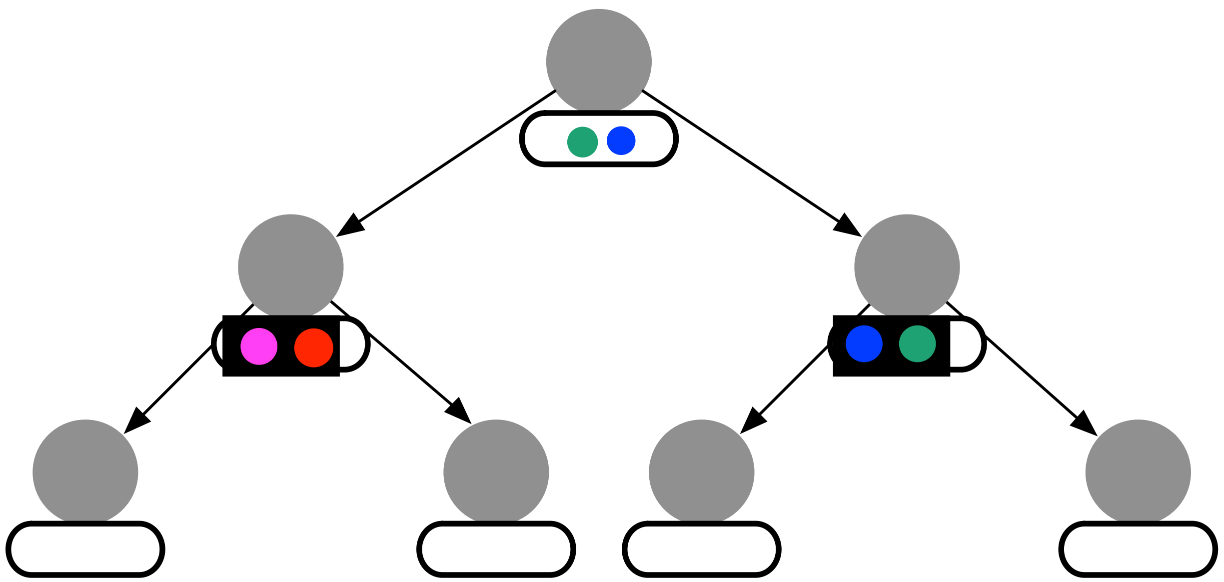
b. merged

c. aggregated

d. spilled

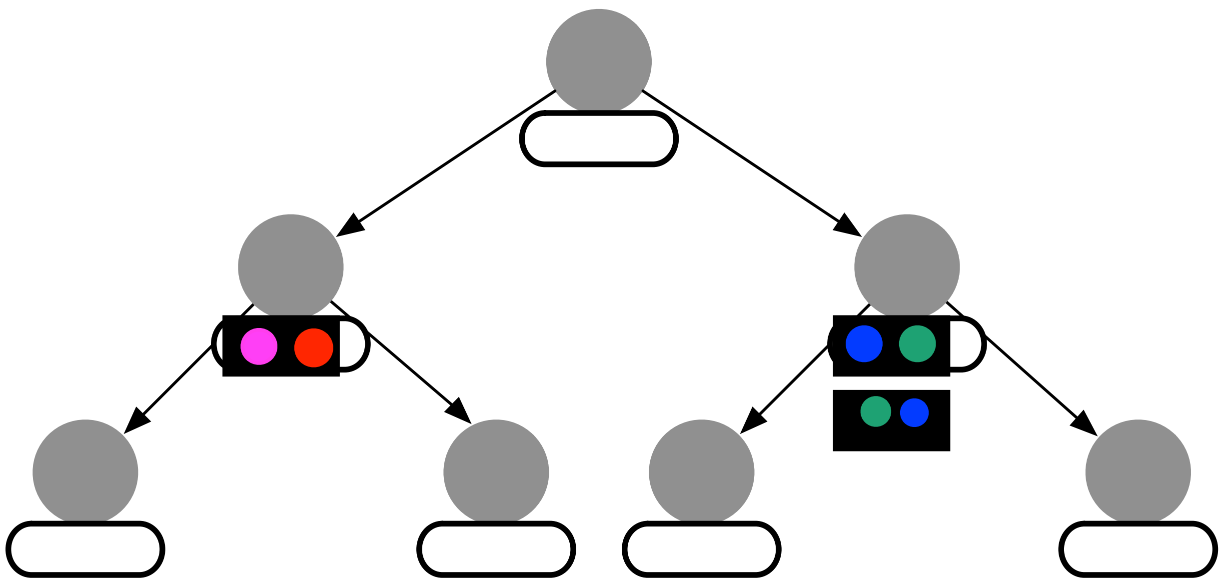




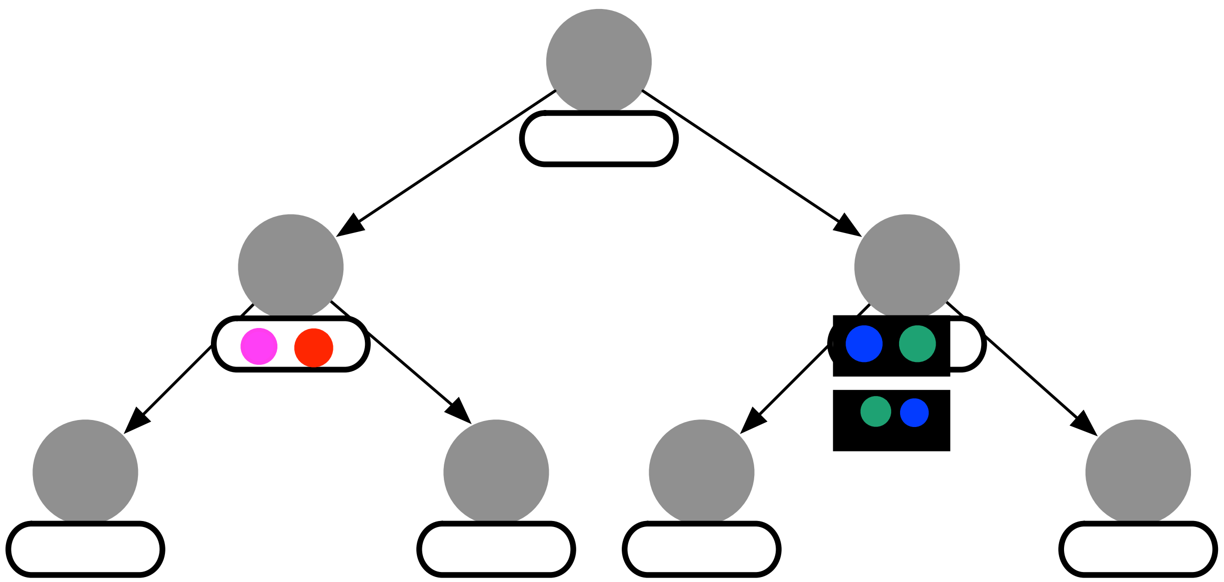


⑥ After all inserts, tree is flushed

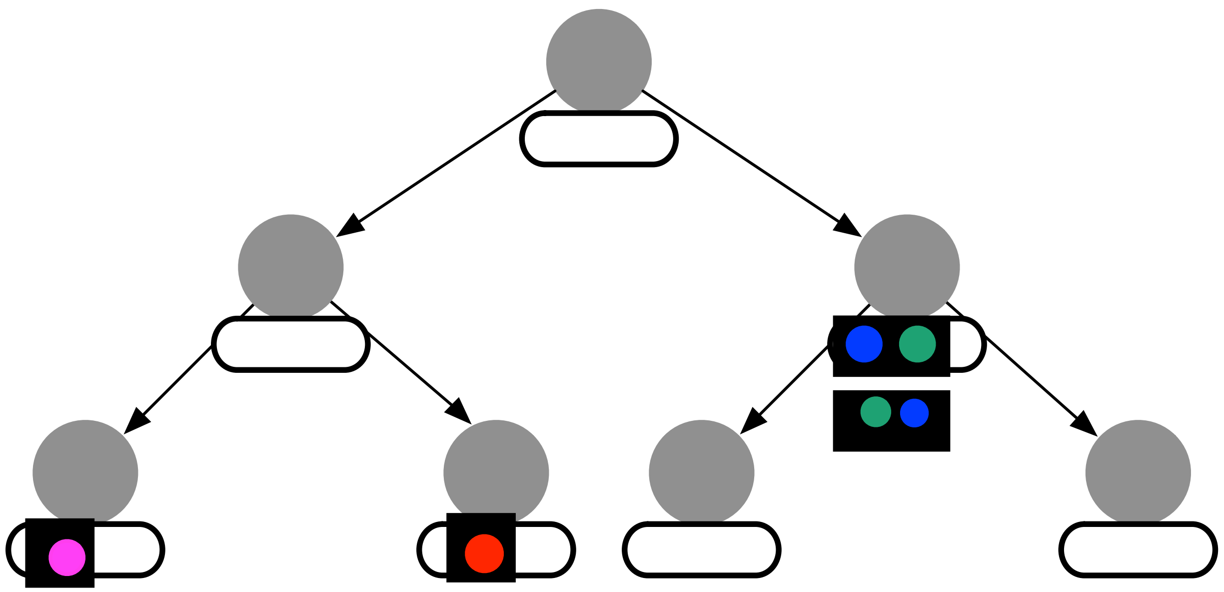
⑥ After all inserts, tree is flushed

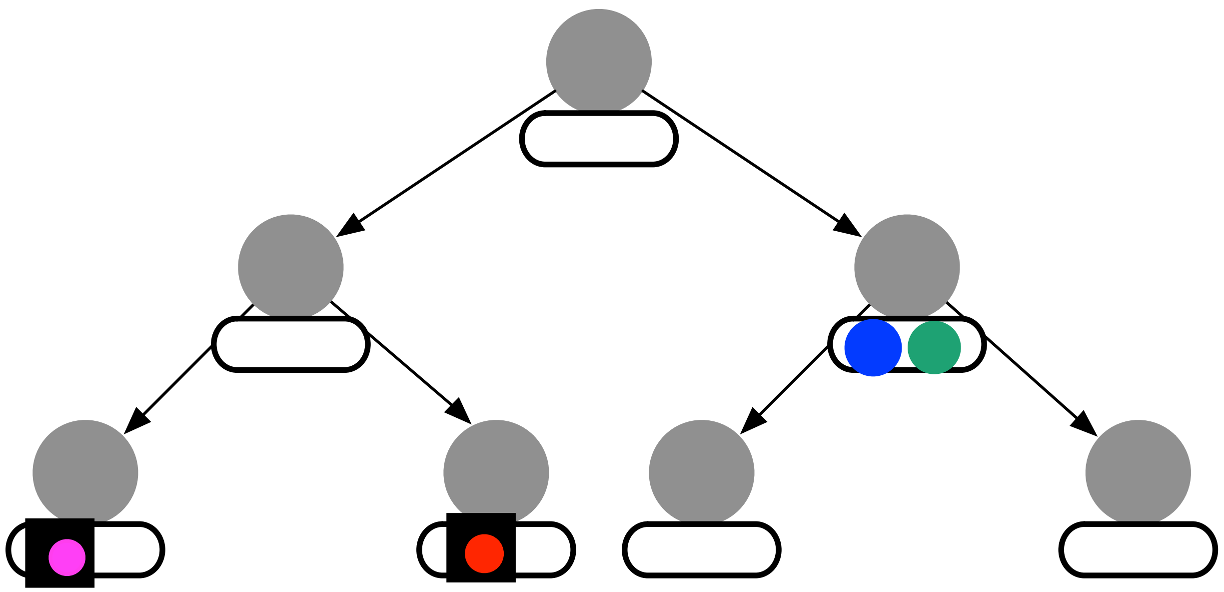


⑥ After all inserts, tree is flushed



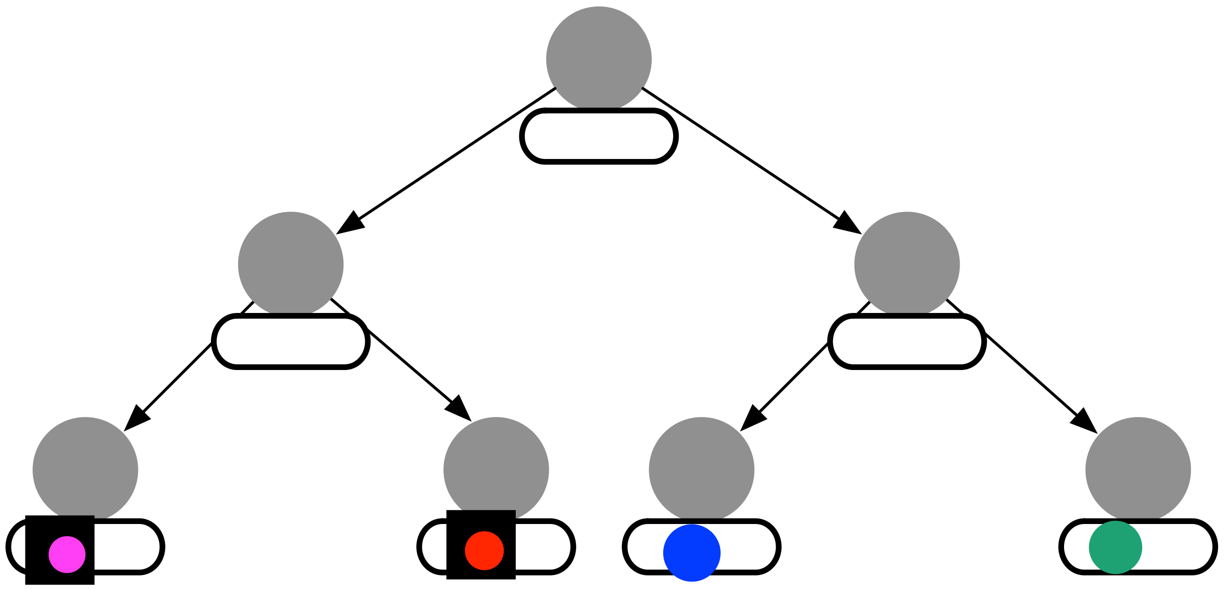
⑥ After all inserts, tree is flushed



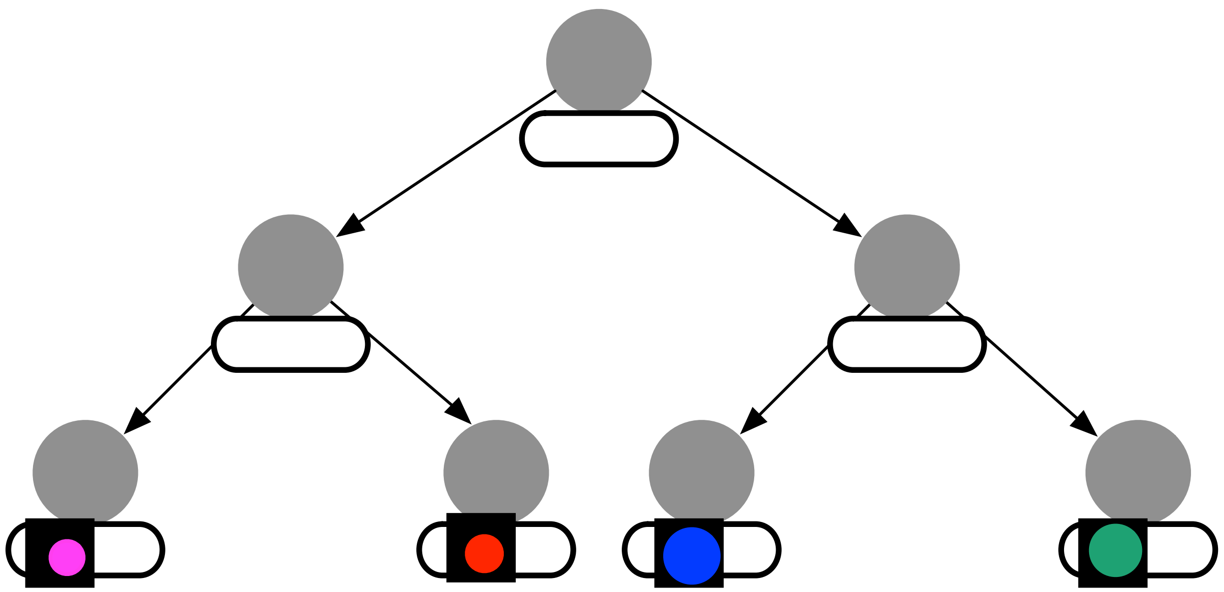


⑥ After all inserts, tree is flushed

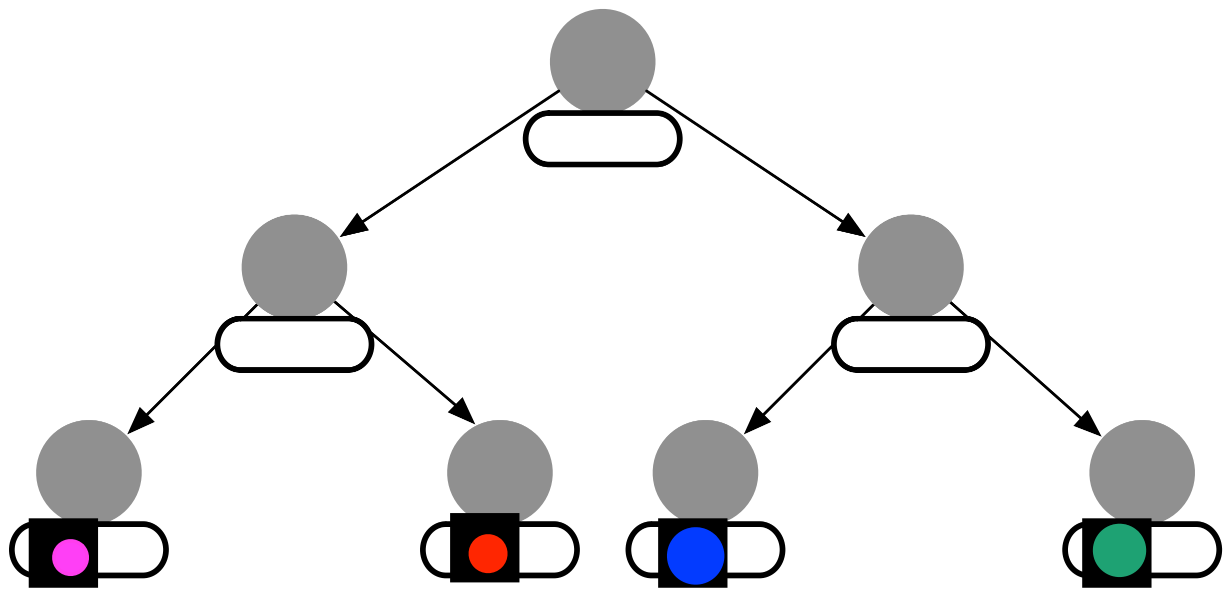
⑥ After all inserts, tree is flushed



⑥ After all inserts, tree is flushed



⑥ After all inserts, tree is flushed



⑦ Aggregated results available in leaves

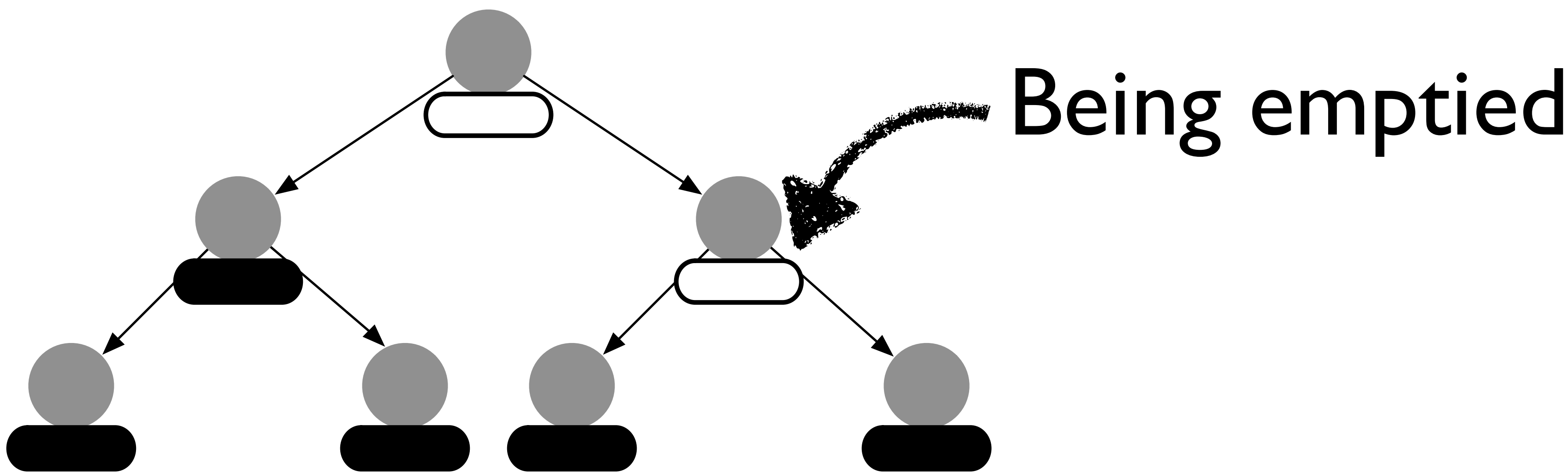
CBT Operation (recap)

- **PAO**s always inserted into root buffer
- If root full, sort **PAO**s, aggregate and spill
- Spilled buffer fragments are compressed in memory
- If child is full, decompress fragments, merge and spill recursively
- Flush tree at the end

Compressed Buffer Tree

Compressed Buffer Tree

Memory efficiency through compression



Compressed Buffer Tree

Memory efficiency through compression

Compressed Buffer Tree

Memory efficiency through compression

Effective compression through use of large buffers

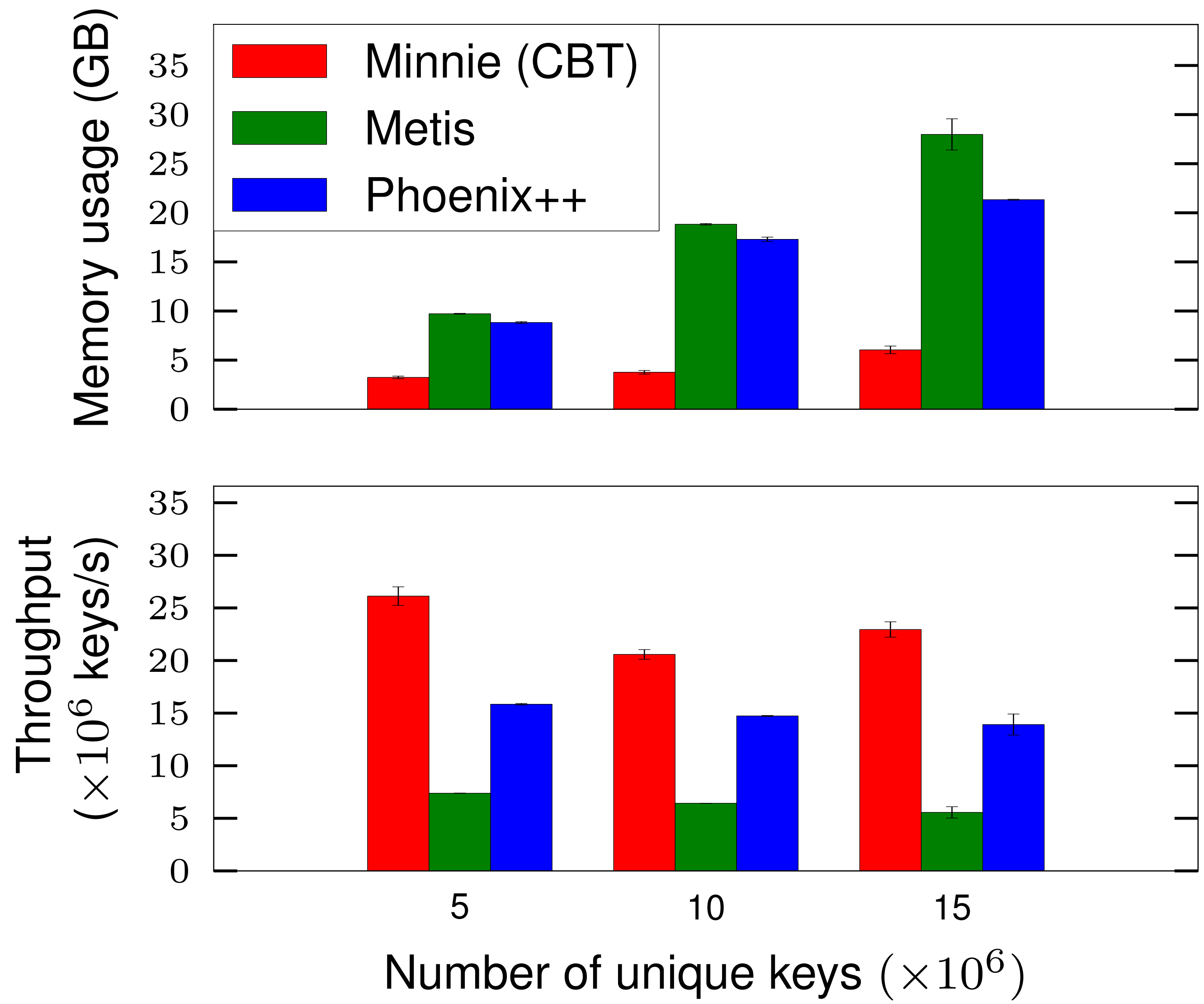
Compressed Buffer Tree

Memory efficiency through compression

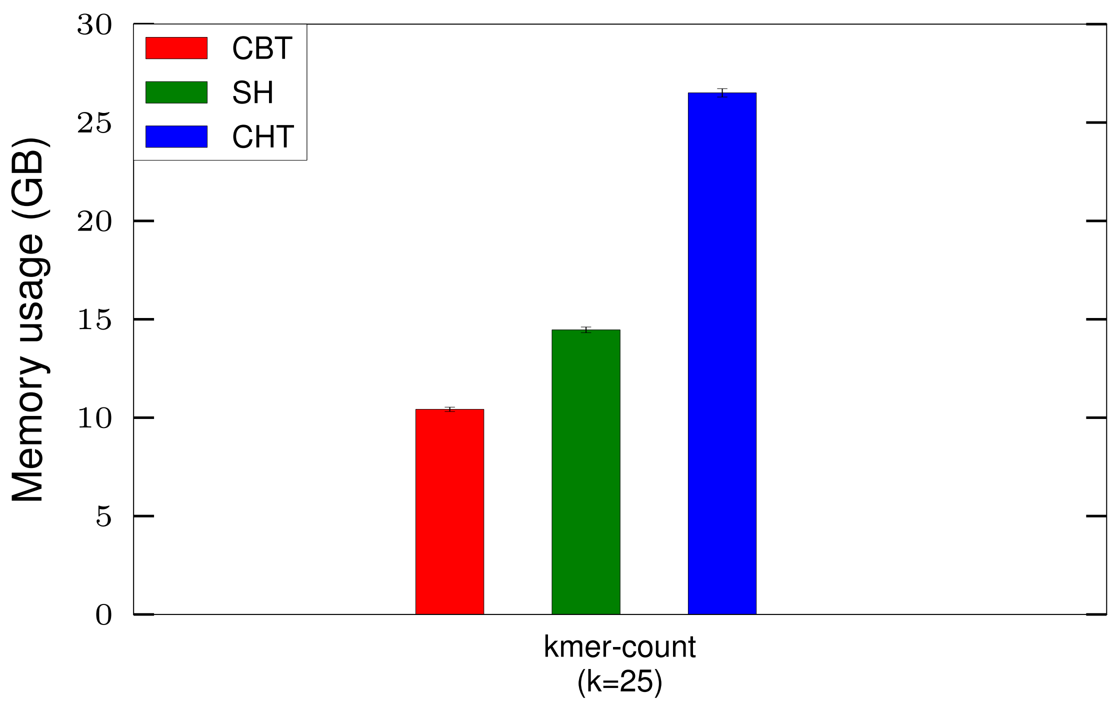
Effective compression through use of large buffers

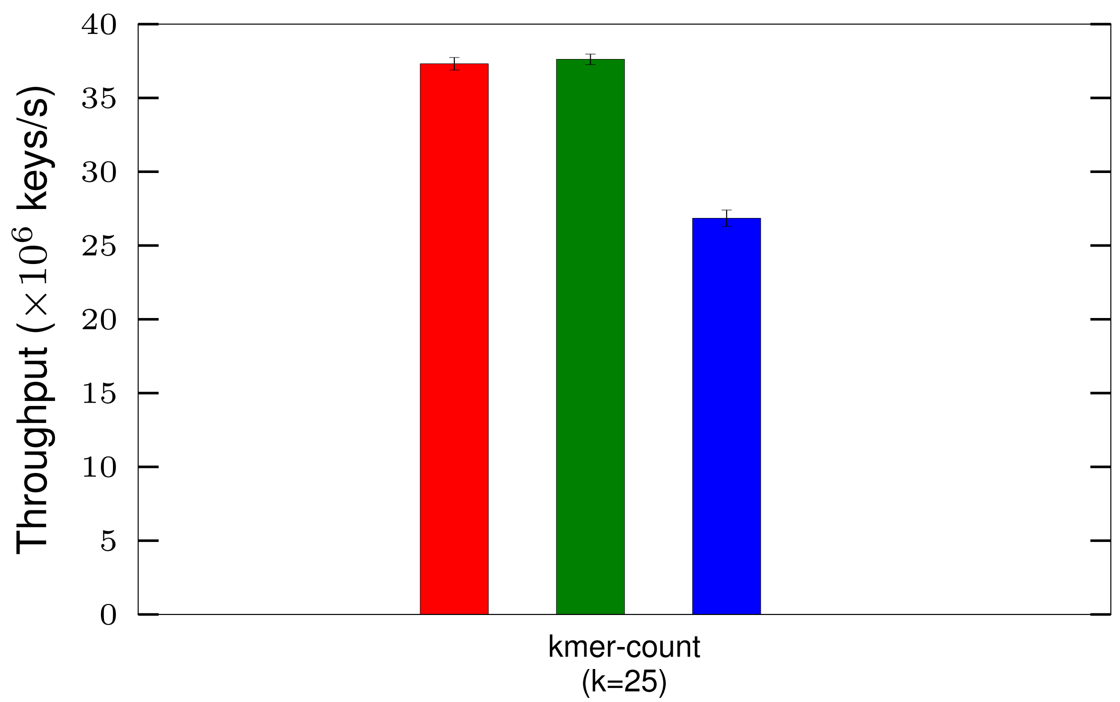
High performance through buffering

Performance



Performance





Thanks!

Memory efficiency through compression

Effective compression through use of large buffers

High performance through buffering