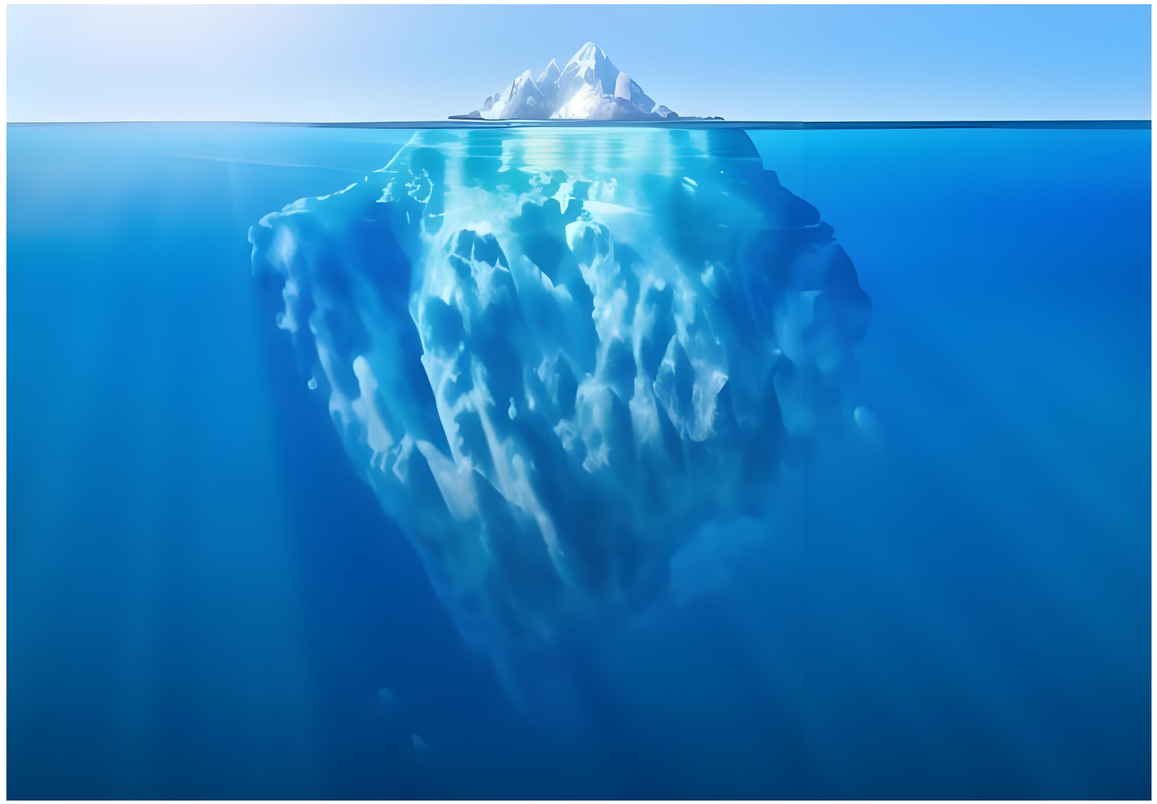
Building a high-performance concurrent map in Go

YunHao Zhang

ByteDance programming language team

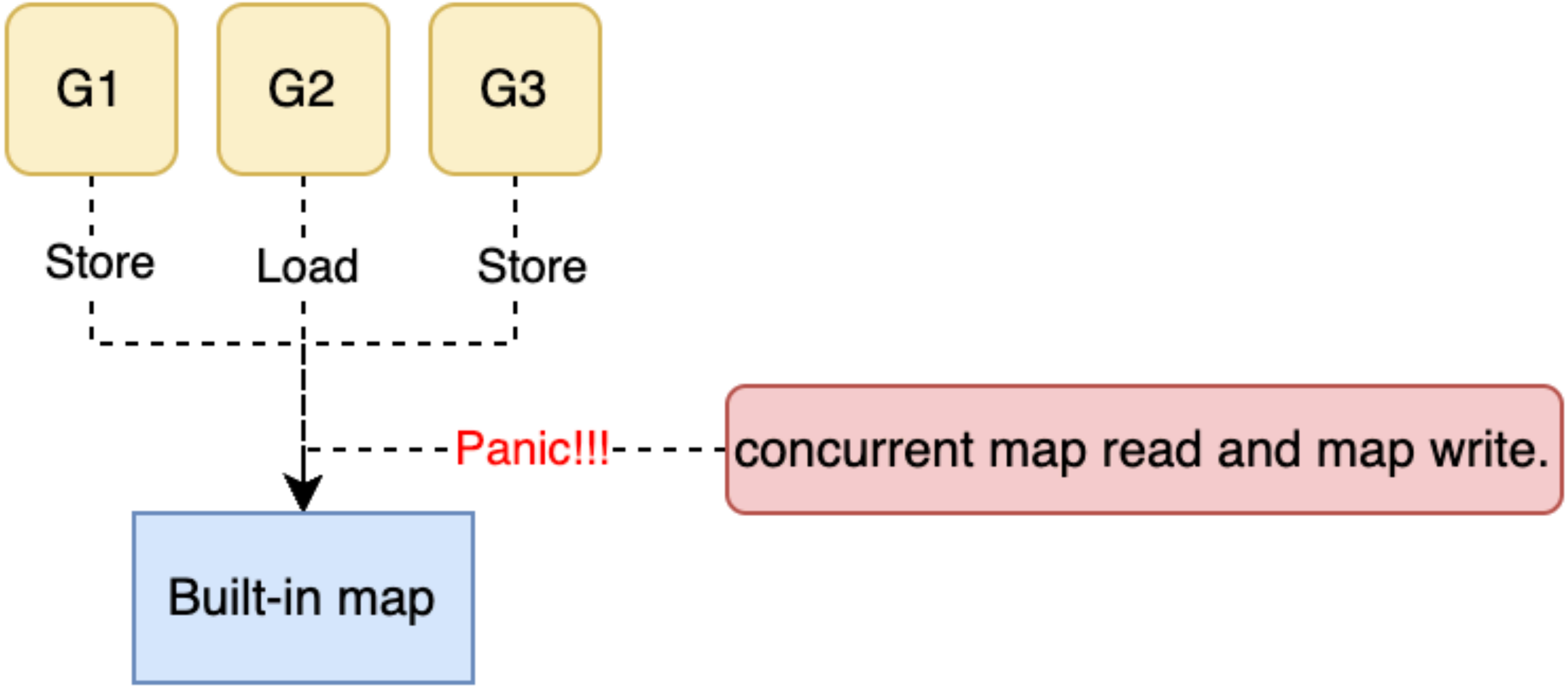
github.com/zhangyunhao116



Key ideas - 25 min

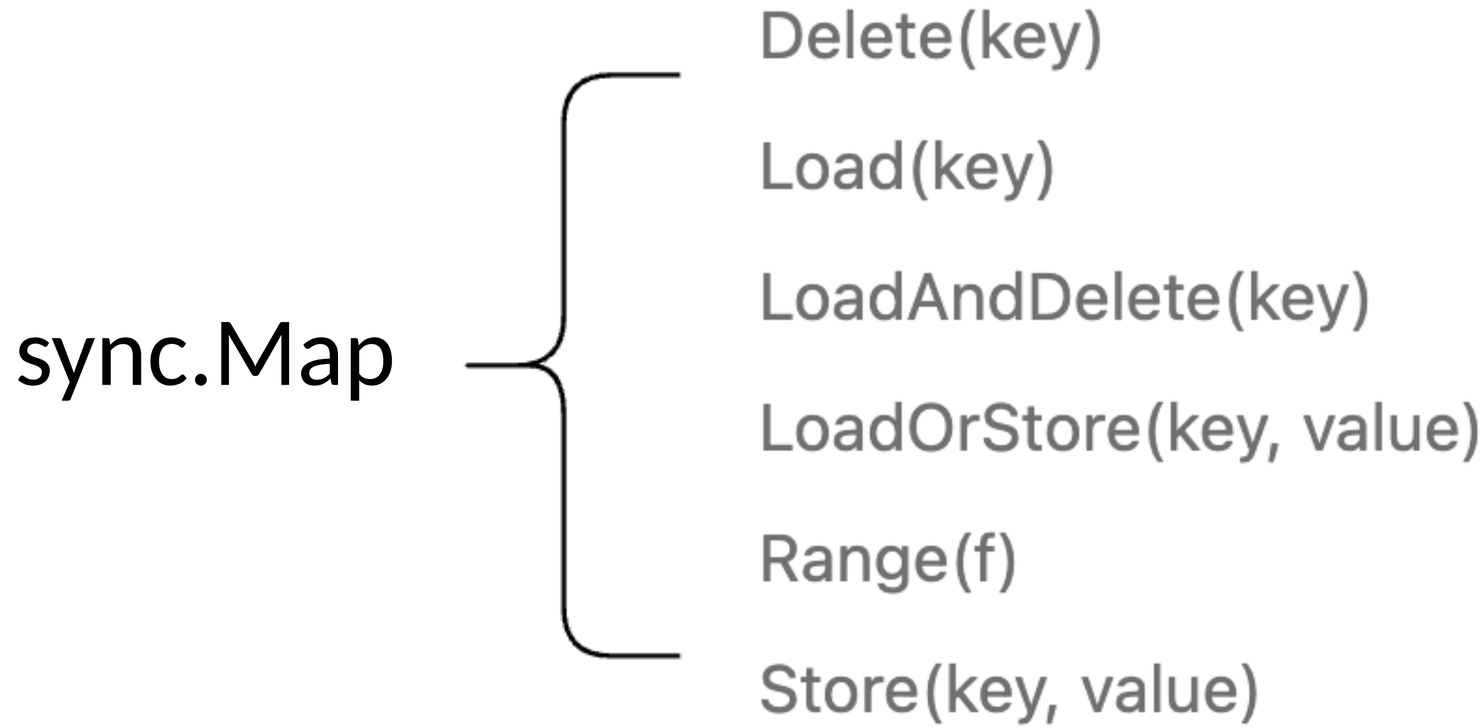
Details - 2h30 min

Why we need concurrent map ?


G1
G2
G3
Store
Load
Store
Panic!!!
concurrent map read and map write.
Built-in map

The built-in map is NOT concurrent-safe

How to build a concurrent-map



map[K]V + sync.RWMutex

	rwmap.txt	syncmap.txt	
	sec/op	sec/op	vs base
LoadSize1000-16	29.565n ± 1%	7.151n ± 23%	-75.81% (p=0.000 n=10)
Store-16	497.0n ± 8%	913.4n ± 3%	+83.77% (p=0.000 n=10)
70Load30Store-16	195.1n ± 5%	699.5n ± 5%	+258.56% (p=0.000 n=10)
90Load9Store1Delete-16	139.7n ± 2%	595.5n ± 1%	+326.42% (p=0.000 n=10)
90Load8Store1Delete1Range-16	8.497µ ± 2%	40.594µ ± 1%	+377.75% (p=0.000 n=10)
geomean	320.9n	643.6n	+100.58%

LoadSize1000: Only Load operation in a map with 1000 items

Store: Only Store operation in a map

70Load30Store: 70% Load and 30% Store operations

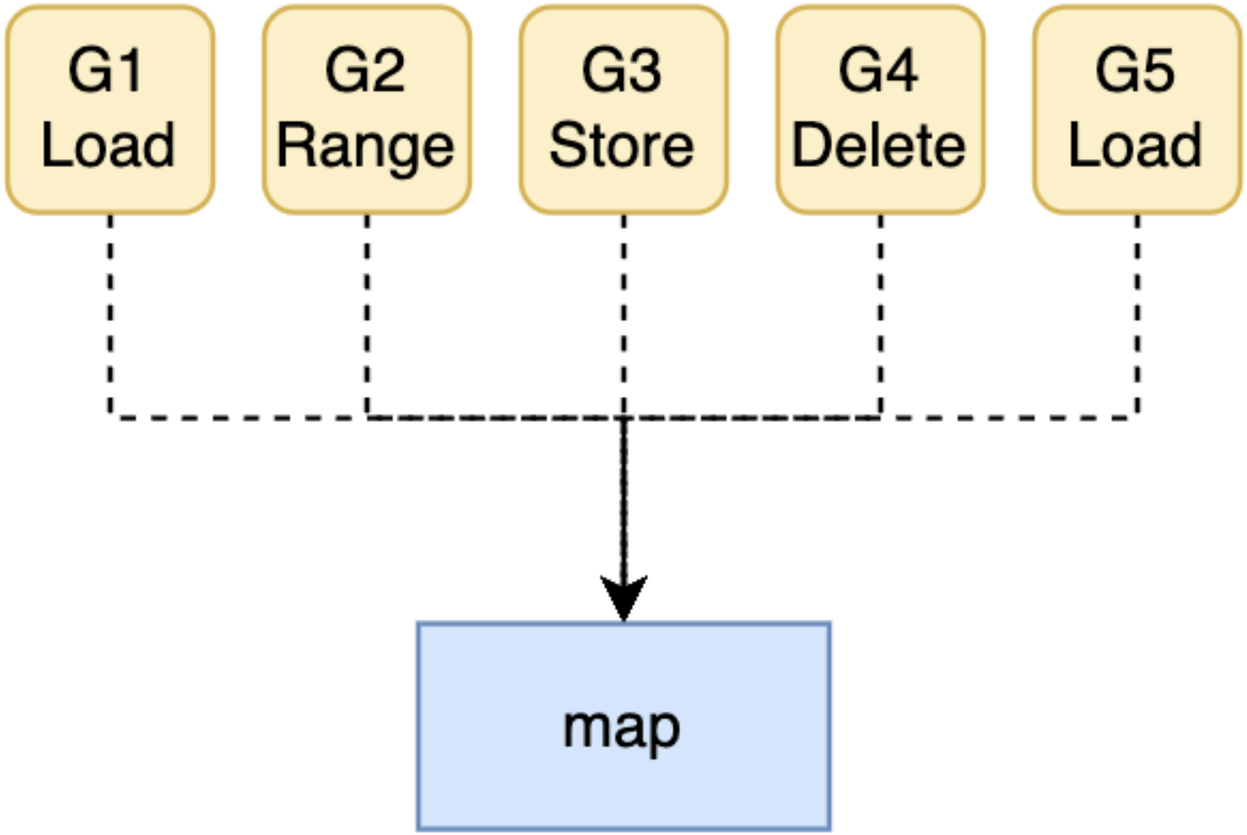
90Load9Store1Delete: 90% Load, 9% Store, 1% Delete

90Load8Store1Delete1Range: 90% Load, 8% Store, 1% Delete, 1% Range

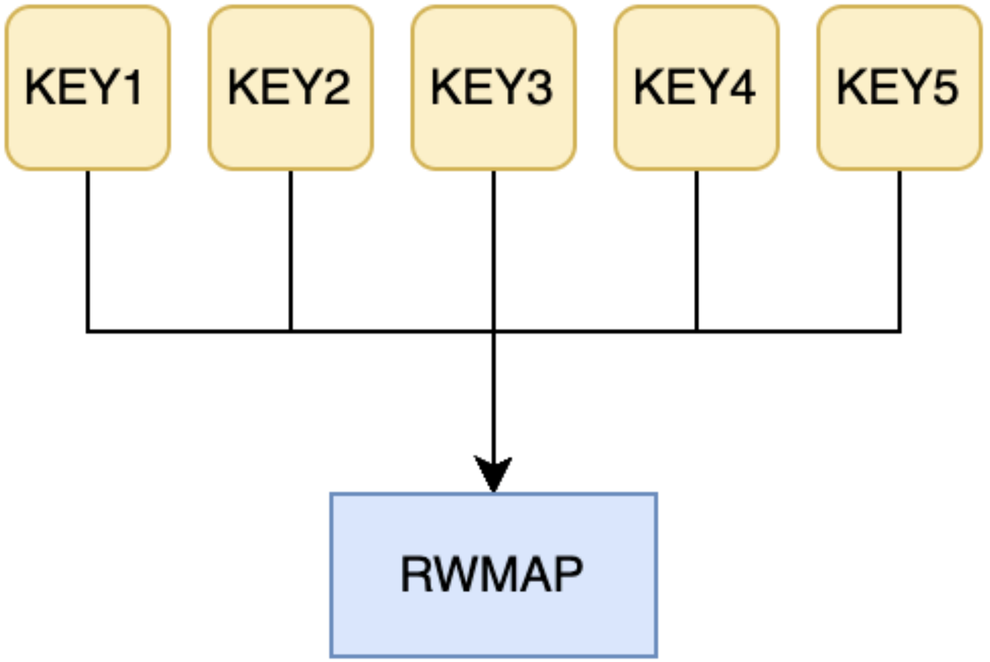
For Load-only case, sync.Map is faster

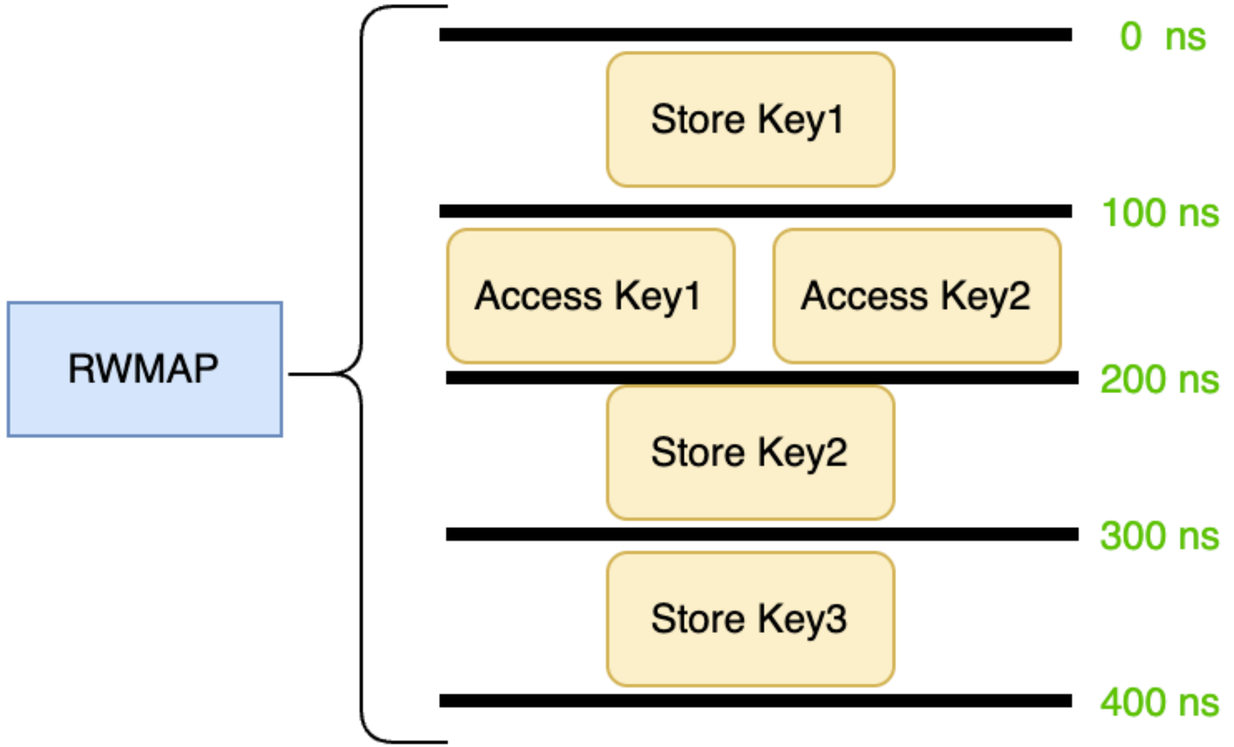
For Mixed read-write cases, read-write mutex map is faster

sync.Map is designed to solve the problem known as cache contention



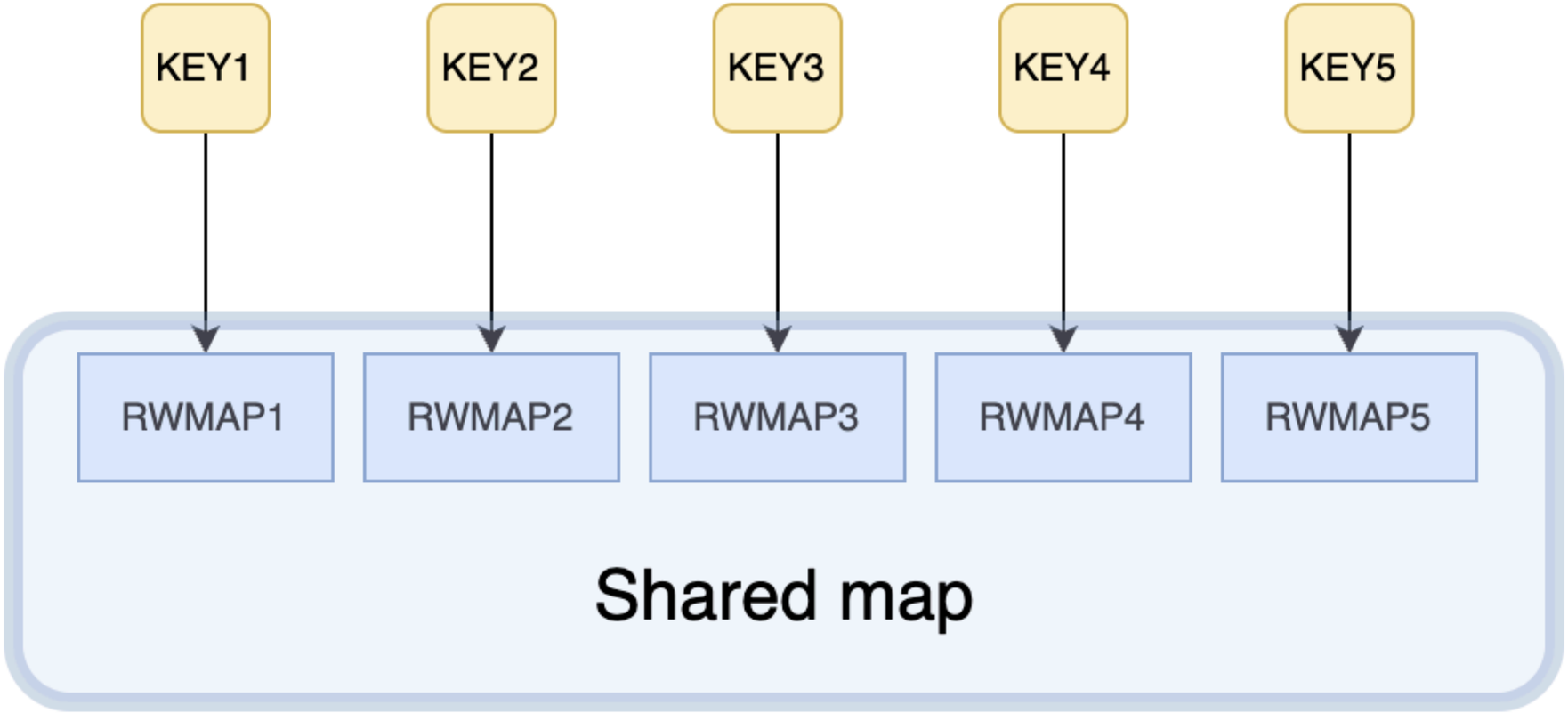
Using the map as a concurrent cache





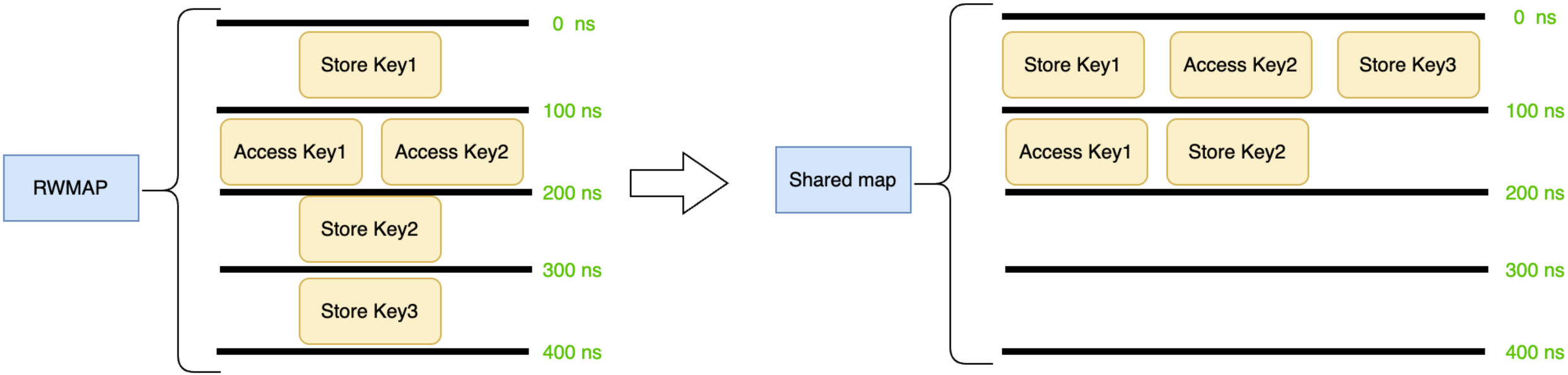
*Only one write operation can be executed at a time

*Multiple read operations can be executed concurrently



Shared map = multiple read-write mutex maps

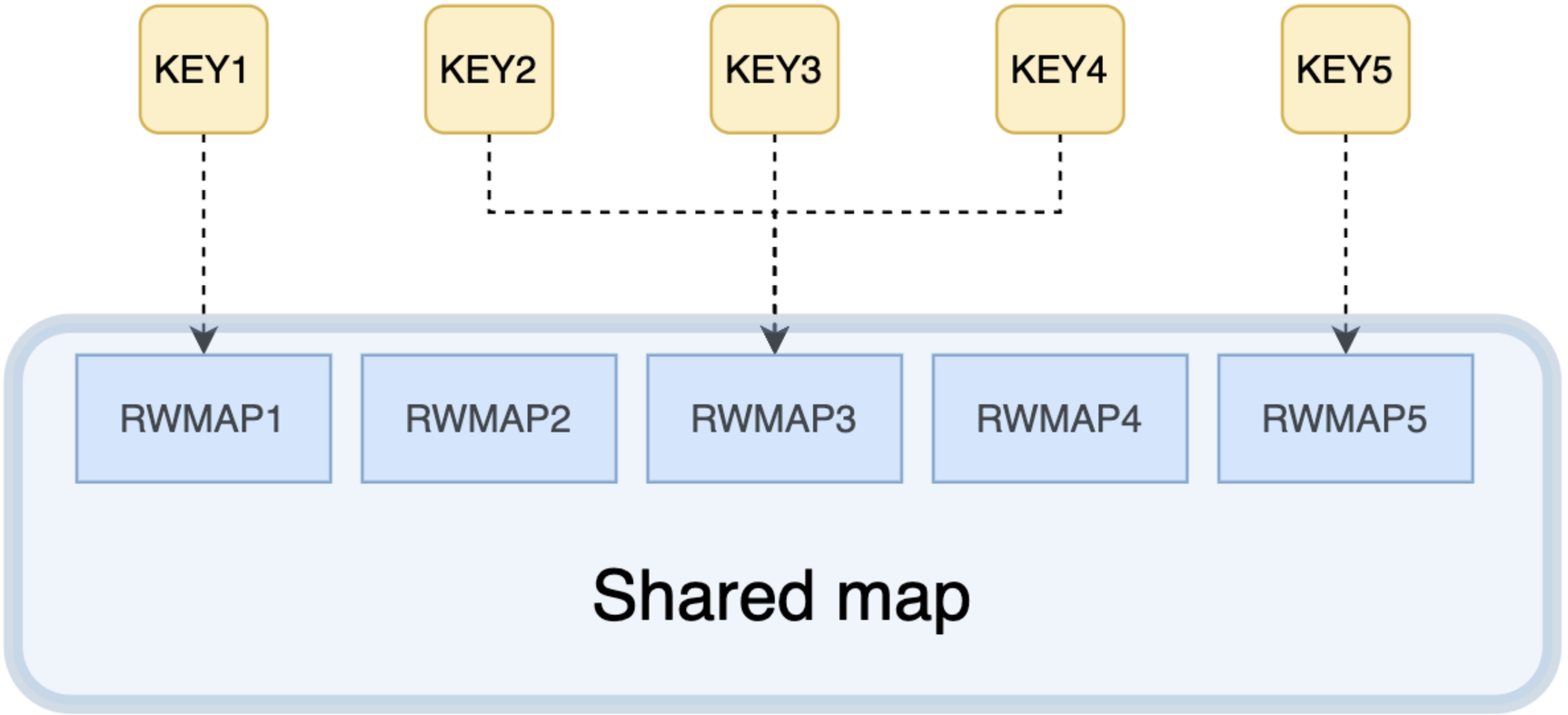
* X CPU cores => X read-write mutex maps in the shared map
* Use a hash function to determine which map a key should go to



The single read-write mutex map versus the shared map.

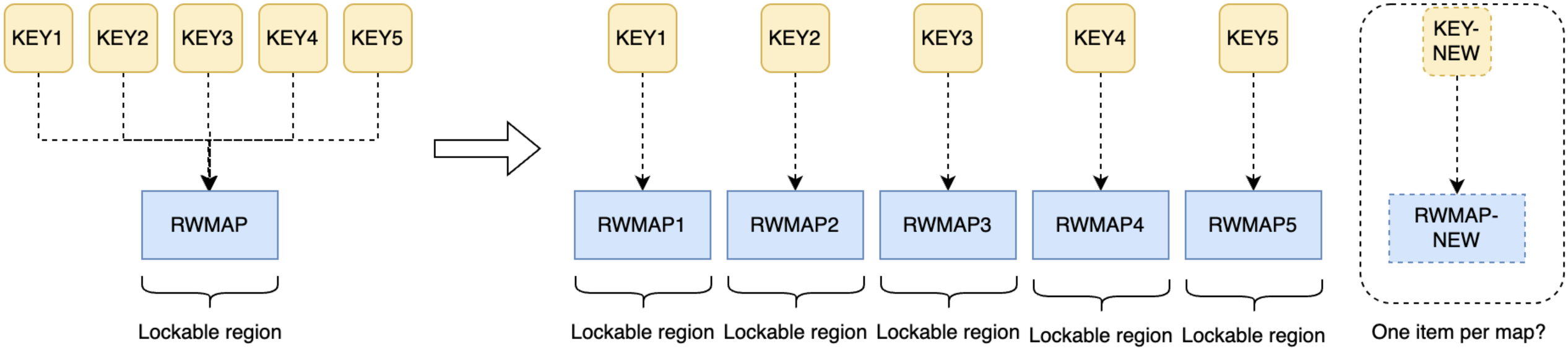
	rwmap.txt	sharedmap.txt	
	sec/op	sec/op	vs base
LoadSize1000-16	29.56n ± 1%	13.63n ± 11%	-53.88% (p=0.000 n=10)
Store-16	497.0n ± 8%	130.2n ± 7%	-73.80% (p=0.000 n=10)
70Load30Store-16	195.1n ± 5%	151.9n ± 2%	-22.17% (p=0.000 n=10)
90Load9Store1Delete-16	139.65n ± 2%	92.69n ± 3%	-33.63% (p=0.000 n=10)
90Load8Store1Delete1Range-16	8.497µ ± 2%	1.652µ ± 2%	-80.56% (p=0.000 n=10)
geomean	320.9n	132.8n	-58.62%

Shared map is faster in all cases, but it requires more memory



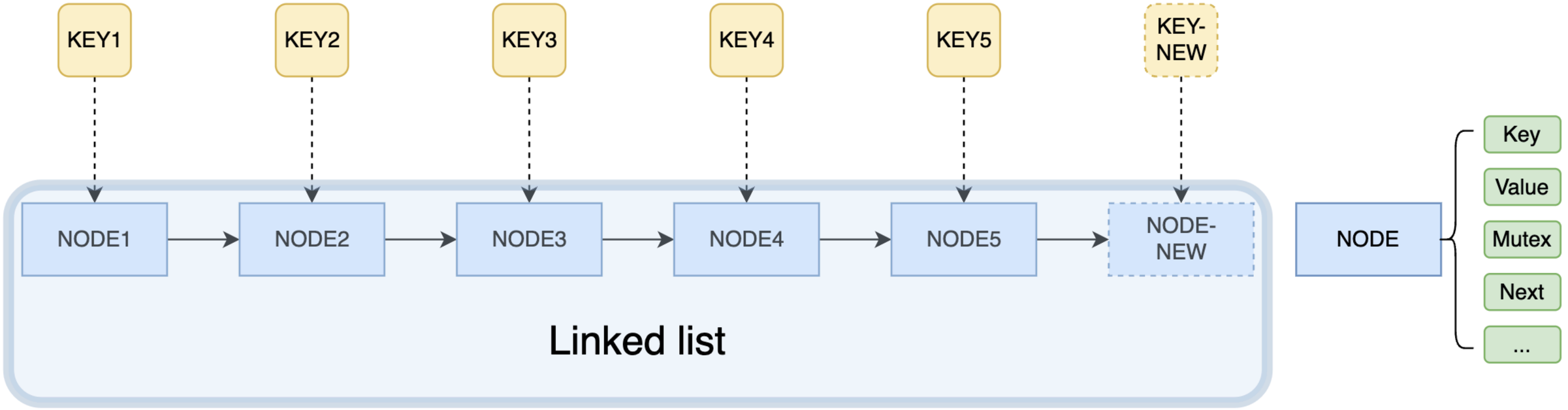
A perfect Hash function doesn’t exist!

(Different operations may still go to the same sub-map)

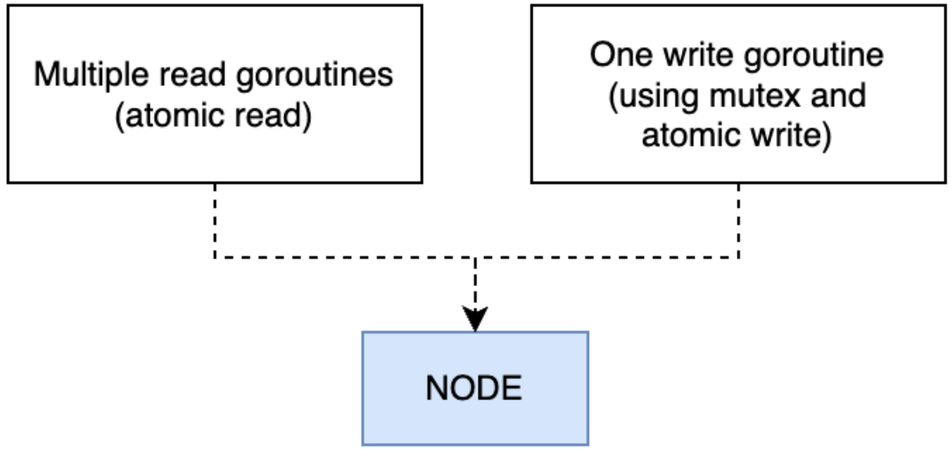


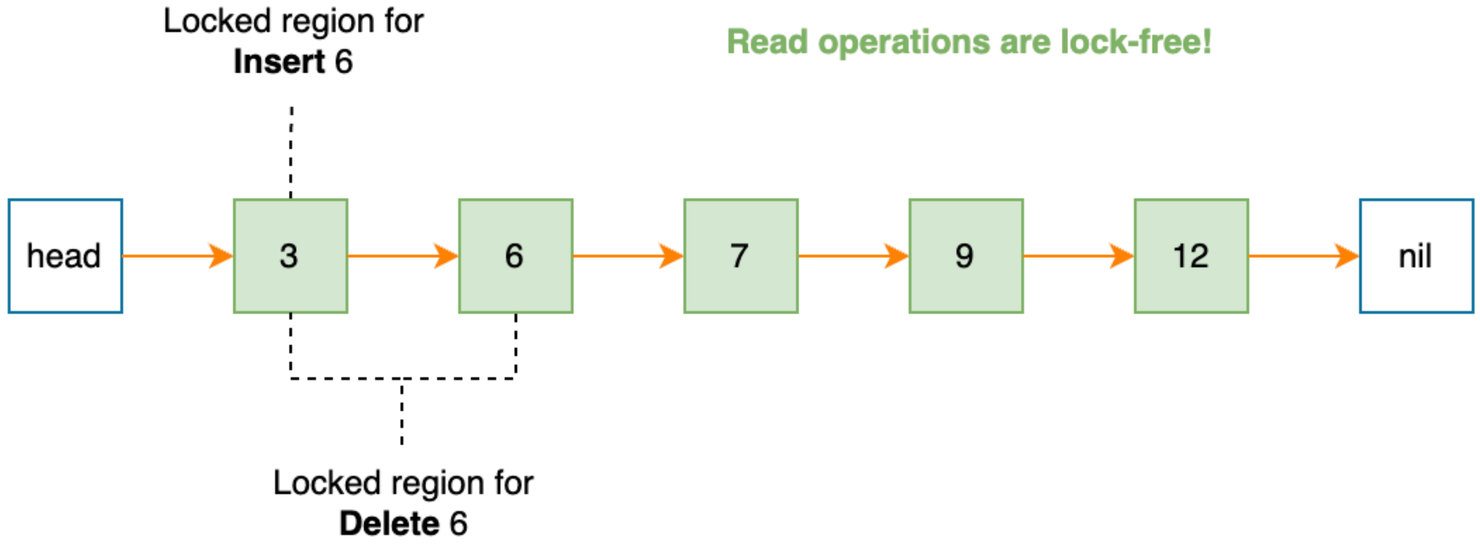
We upgraded a single read-write mutex map to a shared map (multiple read-write mutex maps), and by reducing the locked regions for each write operation, we achieved a 2x performance improvement.

Let’s try one item per sub-map



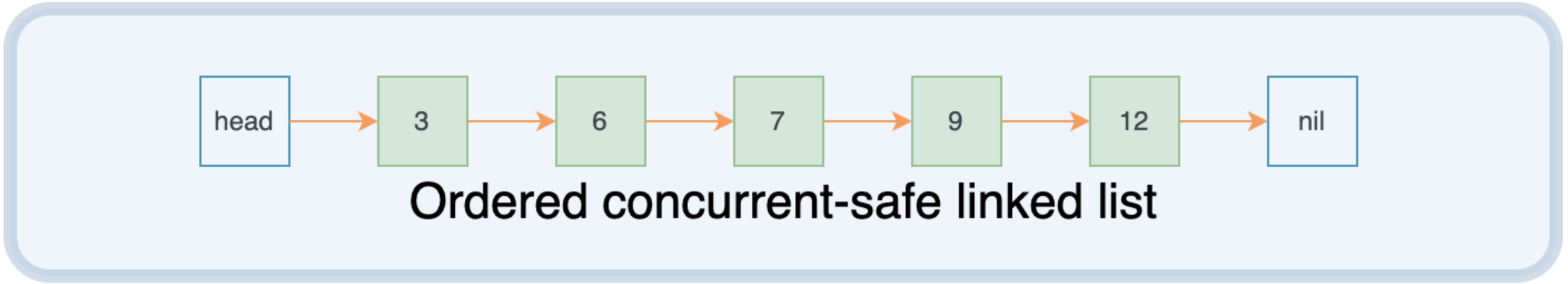
Sub-map (or node, with only one item). Each time a new key-value pair is inserted, we create a new node, and all nodes are connected through pointers, which is actually a linked list.



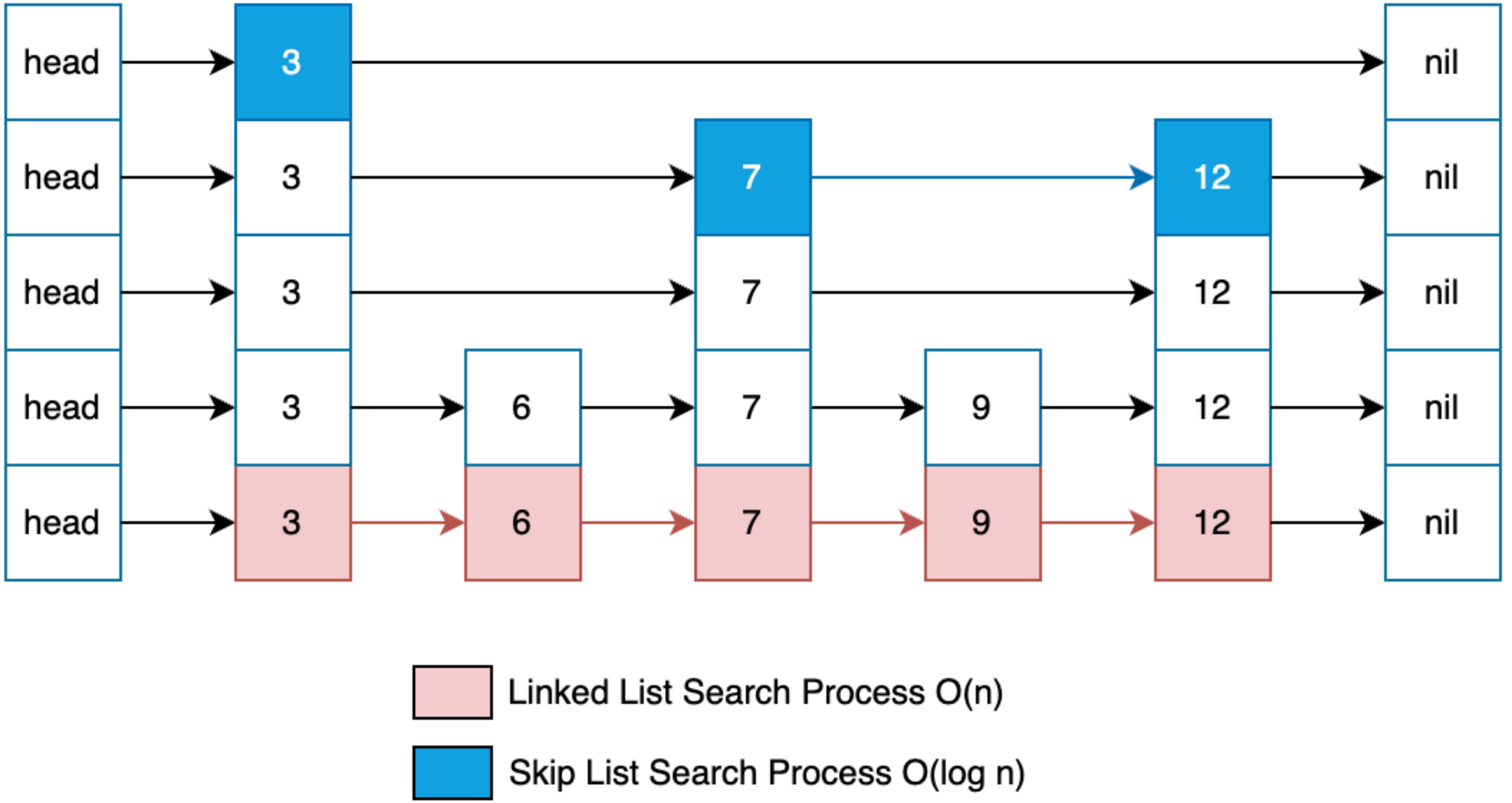


*All write operations must acquire the lock

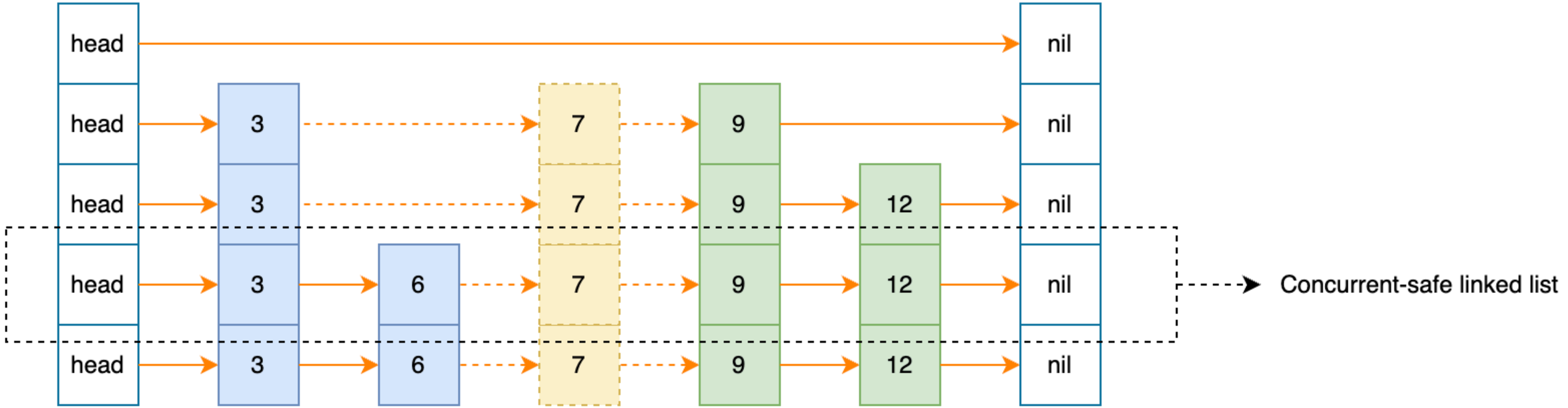
*All read operations are lock-free



Linked list accessing time complexity: O(n)



SkipList = multiple linked lists



Skipmap = multiple concurrent-safe linked lists

	syncmap.txt	skipmap.txt	
	sec/op	sec/op	vs base
LoadSize1000-16	7.151n ± 23%	5.527n ± 15%	-22.71% (p=0.002 n=10)
Store-16	913.4n ± 3%	142.2n ± 5%	-84.44% (p=0.000 n=10)
70Load30Store-16	699.55n ± 5%	62.84n ± 6%	-91.02% (p=0.000 n=10)
90Load9Store1Delete-16	595.50n ± 1%	32.10n ± 5%	-94.61% (p=0.000 n=10)
90Load8Store1Delete1Range-16	40594.0n ± 1%	397.3n ± 2%	-99.02% (p=0.000 n=10)
geomean	643.6n	57.52n	-91.06%

skipmap is ~20x faster in typical case(90%Load9%Store1%Delete)

	Time complexity (read and write)
skipmap	O(log n)
sync.Map	O(1)

skipmap is faster in high-concurrency read-write cases

sync.Map may be faster in low-concurrency cases

Reduce the locked regions for each concurrent operation

* Read-write mutex map -> entire map
* Shared map -> one sub-map
* Skipmap -> few items