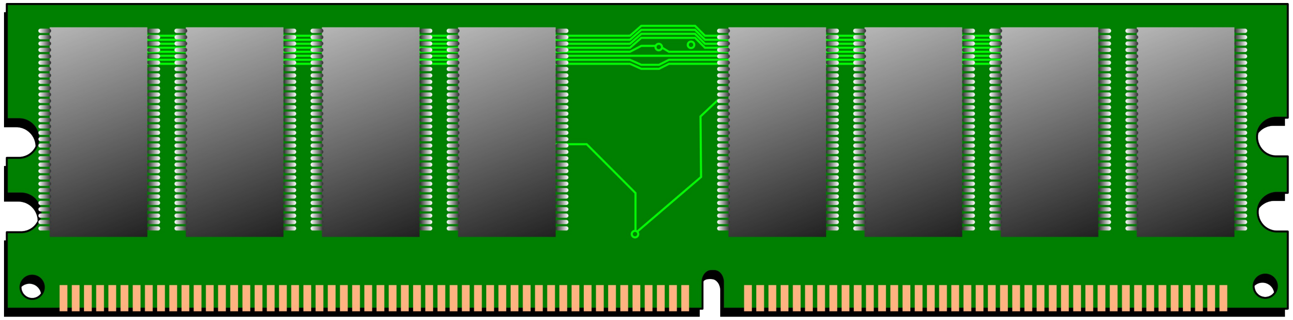
Scalable ETS: Does Such a Thing Exist?

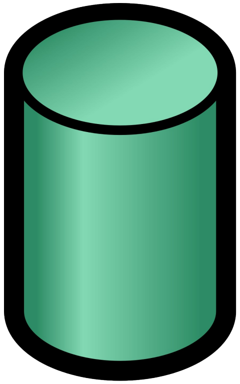
David Klaftenegger
Kjell Winblad

Uppsala University

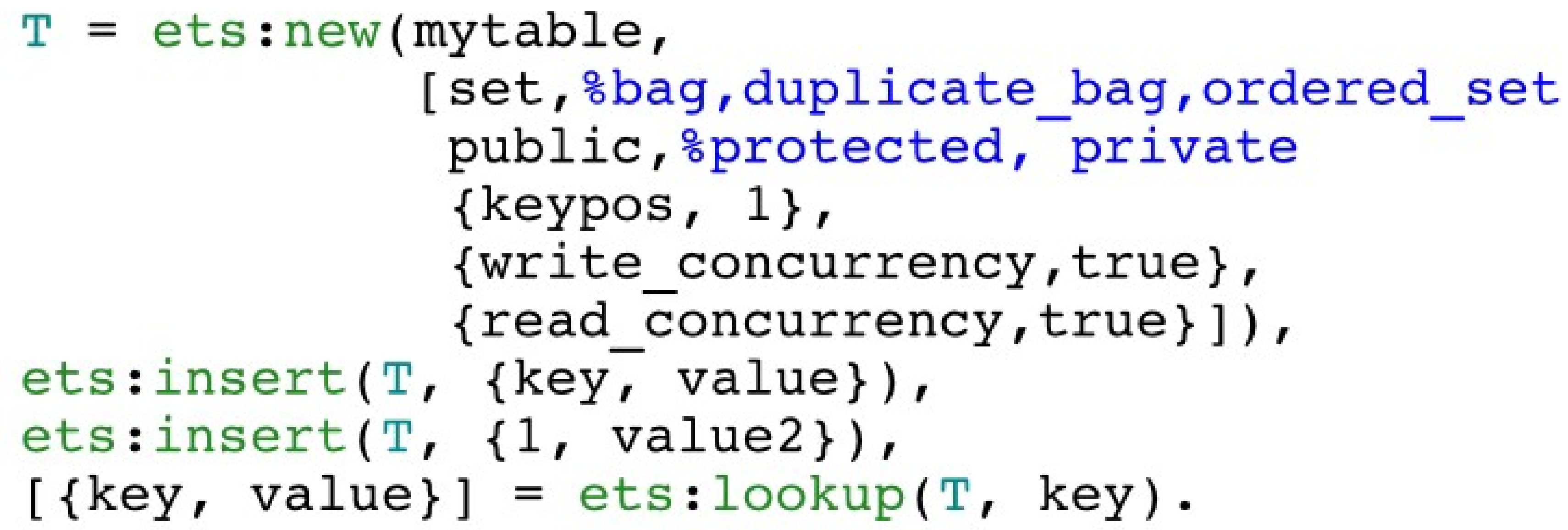
What is ETS?

- Erlang Term Storage
- Key-value store
- lookup/insert/delete, pattern matching queries
- Used by Mnesia
- In-memory database tables





Example



```
T = ets:new(mytable,
            [set,%bag,duplicate_bag,ordered_set
             public,%protected, private
             {keypos, 1},
             {write_concurrency,true},
             {read_concurrency,true}]),
ets:insert(T, {key, value}),
ets:insert(T, {1, value2}),
[{key, value}] = ets:lookup(T, key).
```

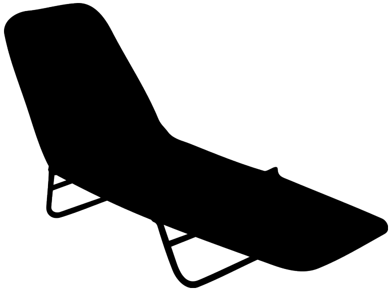
ETS is important

- 86 out of 190 Erlang open source projects had at least one reference to ETS
  - At least 41 use shared tables
  - Libraries not counted

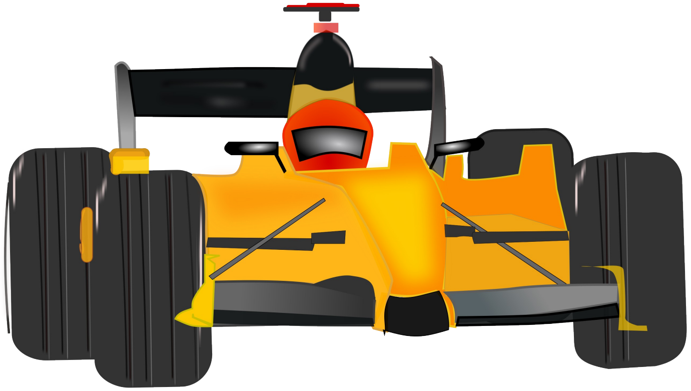
    (if source code was not included)

Why is ETS popular?

- Convenience
  - Provides frequently used functionality in a standard way
  - Easy to use



- Performance
  - Implemented in C
  - Mutable data
  - Scalable?

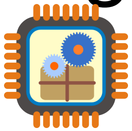


Erlang runtime: communication

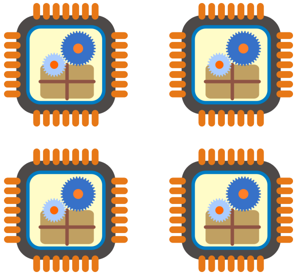
- How to communicate between processes
  - Message passing
    - A processes can only process one message at a time
    - For some applications: serialization point
    - Example application: a cache
  - Writing and reading to shared ETS table
    - Can be done in parallel
    - Or can it?

SMP and NUMA

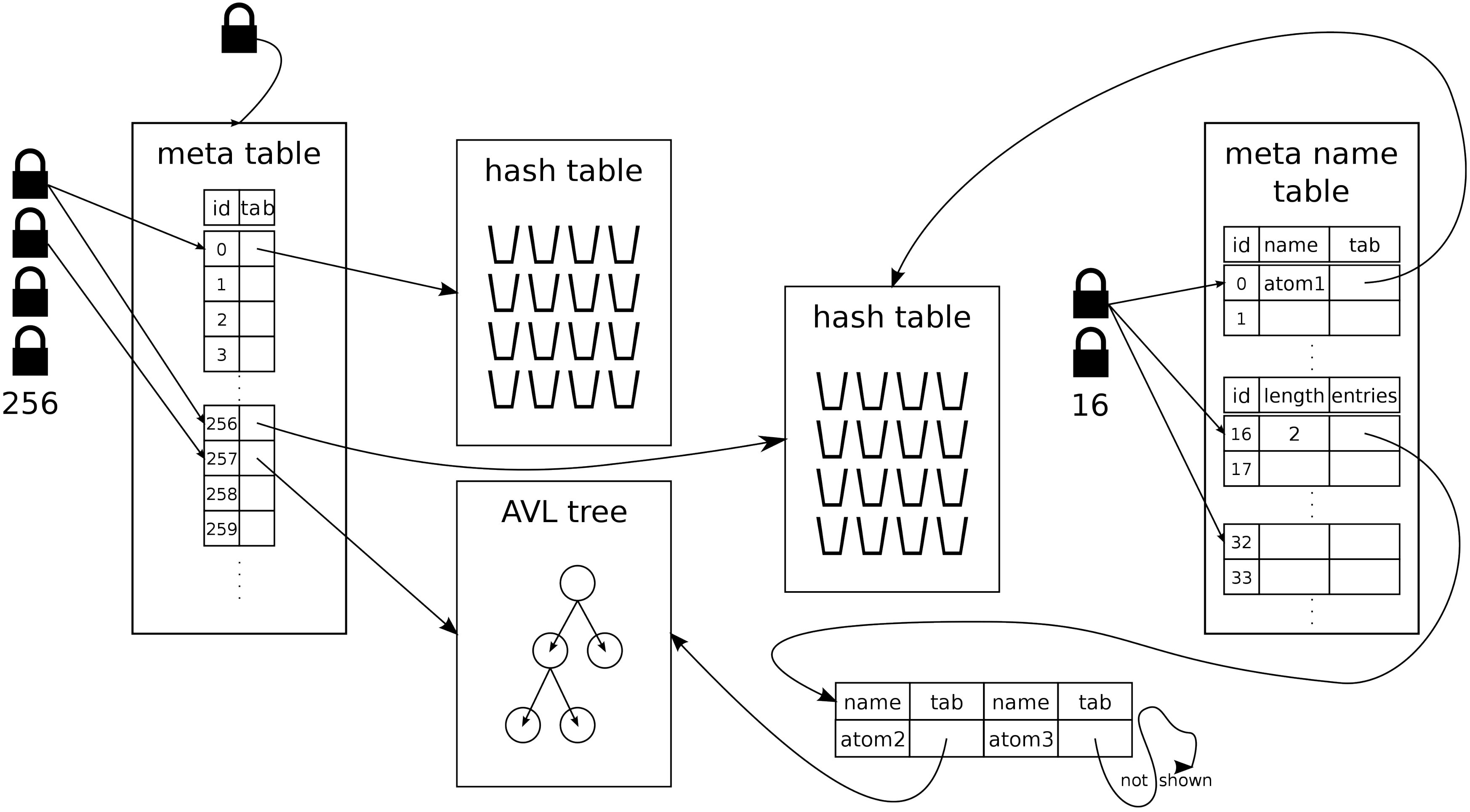
- SMP = Symmetric multiprocessing
  - One chip – multiple cores
  - Communication relatively cheap ☺



- NUMA = Non-Uniform Memory Access
  - Multiple chips – separate memory channels
  - Access to other chip (“remote”) more expensive ☹



ETS Under the Hood

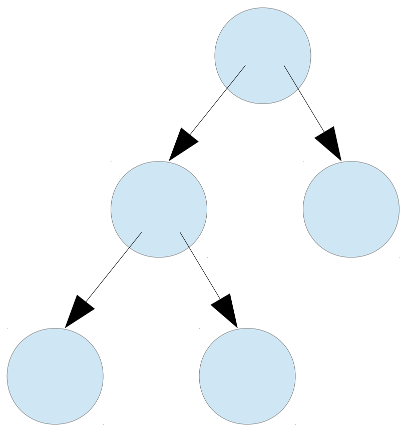


Backend Data Structures

- AVL-tree
  - ordered_set
- Linear Hash Table
  - set
  - bag
  - duplicate_bag

AVL-tree

- Balanced binary search tree
- Protected by single reader-writer lock



For details:
An algorithm for the organization of information(1962)
By:
Adelson-Velskii, G.; E. M. Landis
Published in:
Proceedings of the USSR Academy of Sciences

Linear hash table

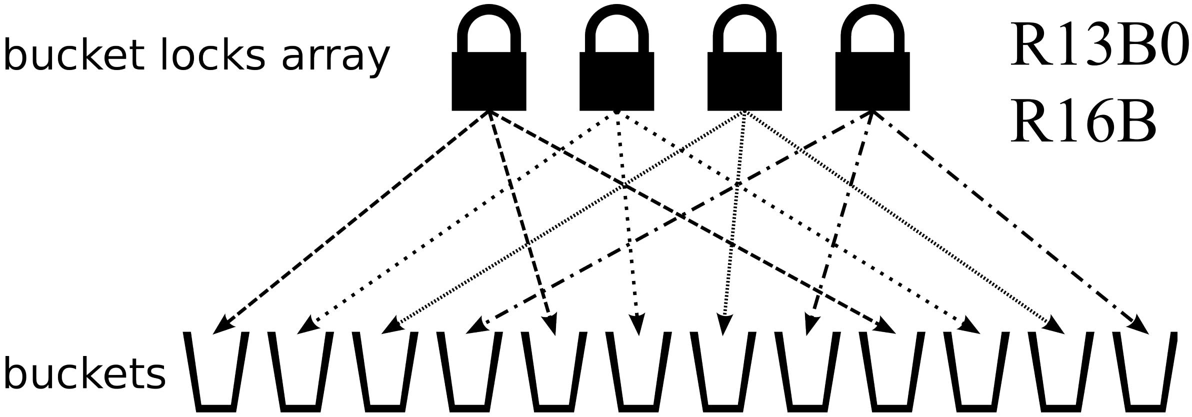
- Hash table
  - Hash key to bucket
  - bucket lists
- Resizing
  - one bucket at a time
- Average bucket length
  - 6 in R16B

For details:
Linear hashing with partial expansions
By:
Per-Åke Larson
Published in:
VLDB '80

Linear hash table

- One reader-writer table lock
- Supports fine-grained locking
- Some operations need full locking anyways
  - Example: insert all elements in a list atomically

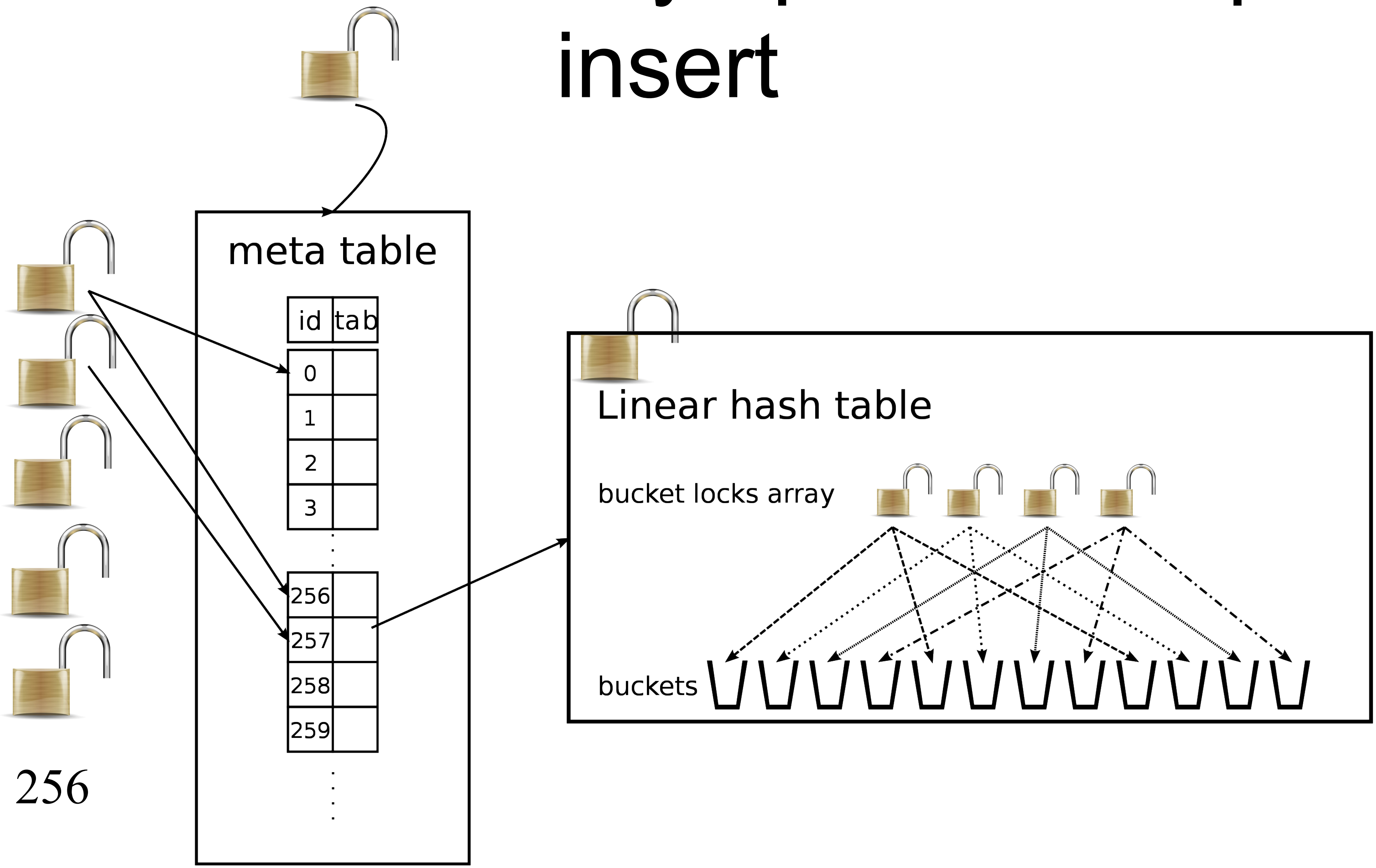
Write concurrency option



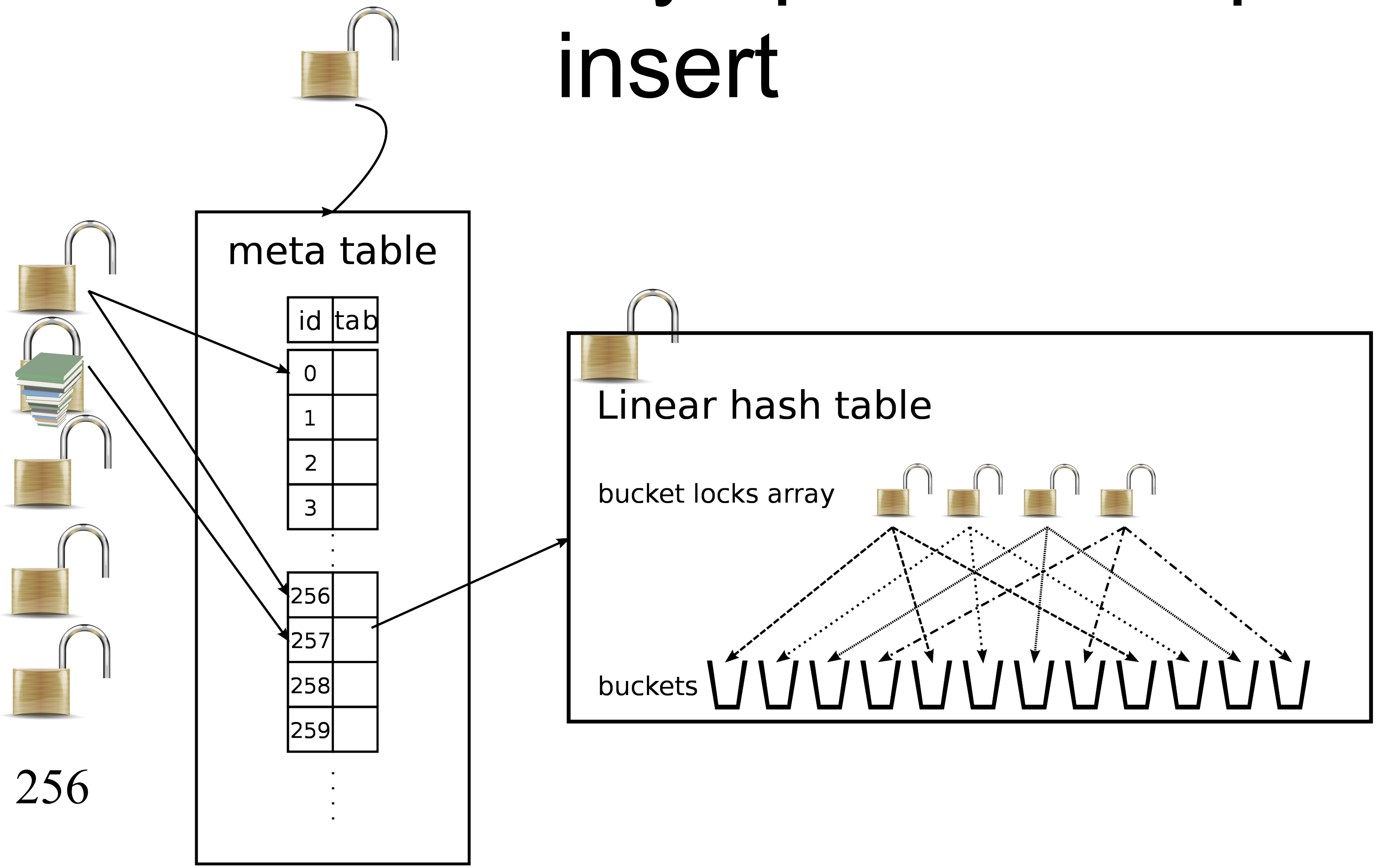
R13B02-1	16 locks
R16B	64 locks

- {write_concurrency, true}
- Introduced in R13B02-1
- Activates array of reader-writer locks

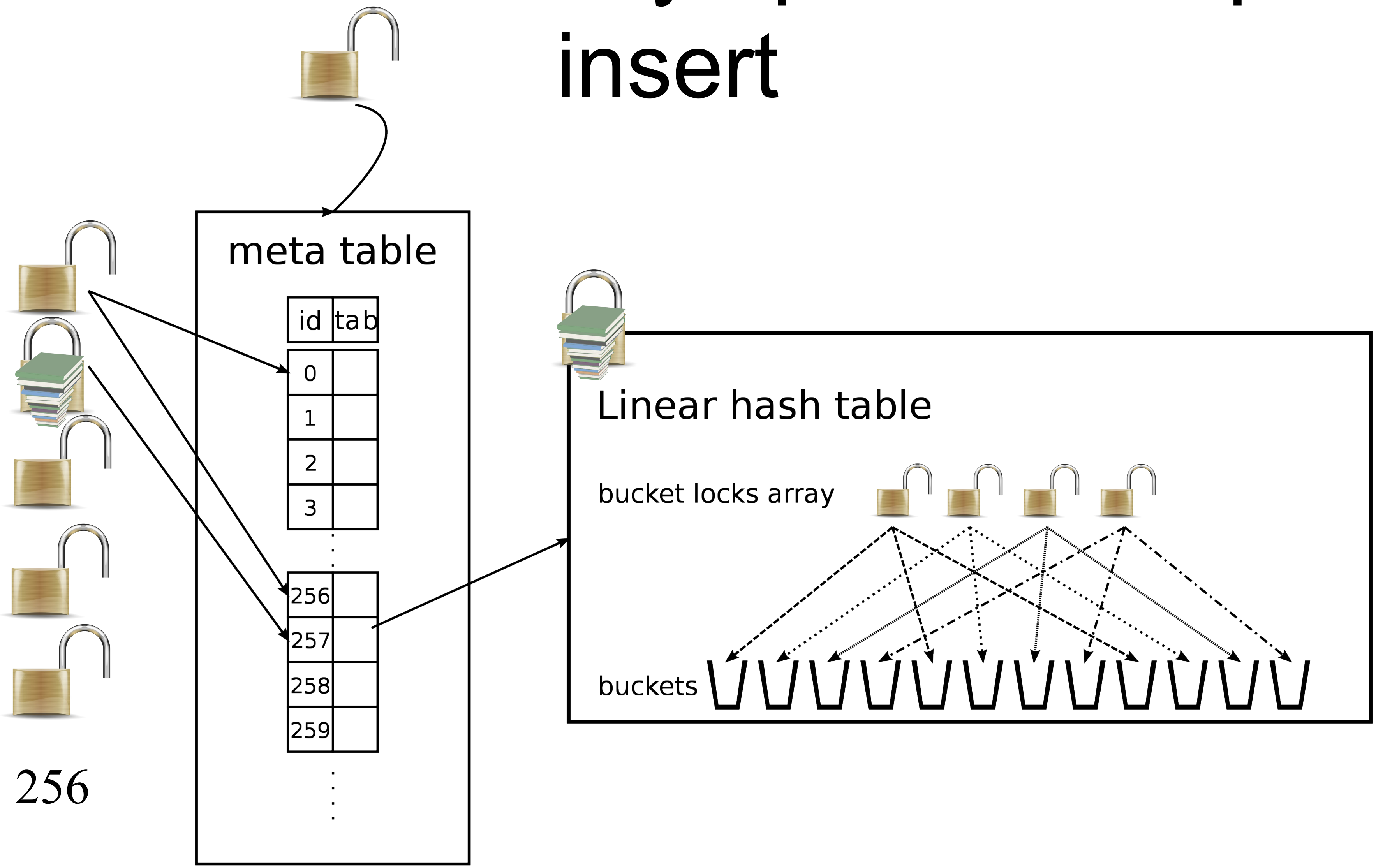
Write concurrency option example insert



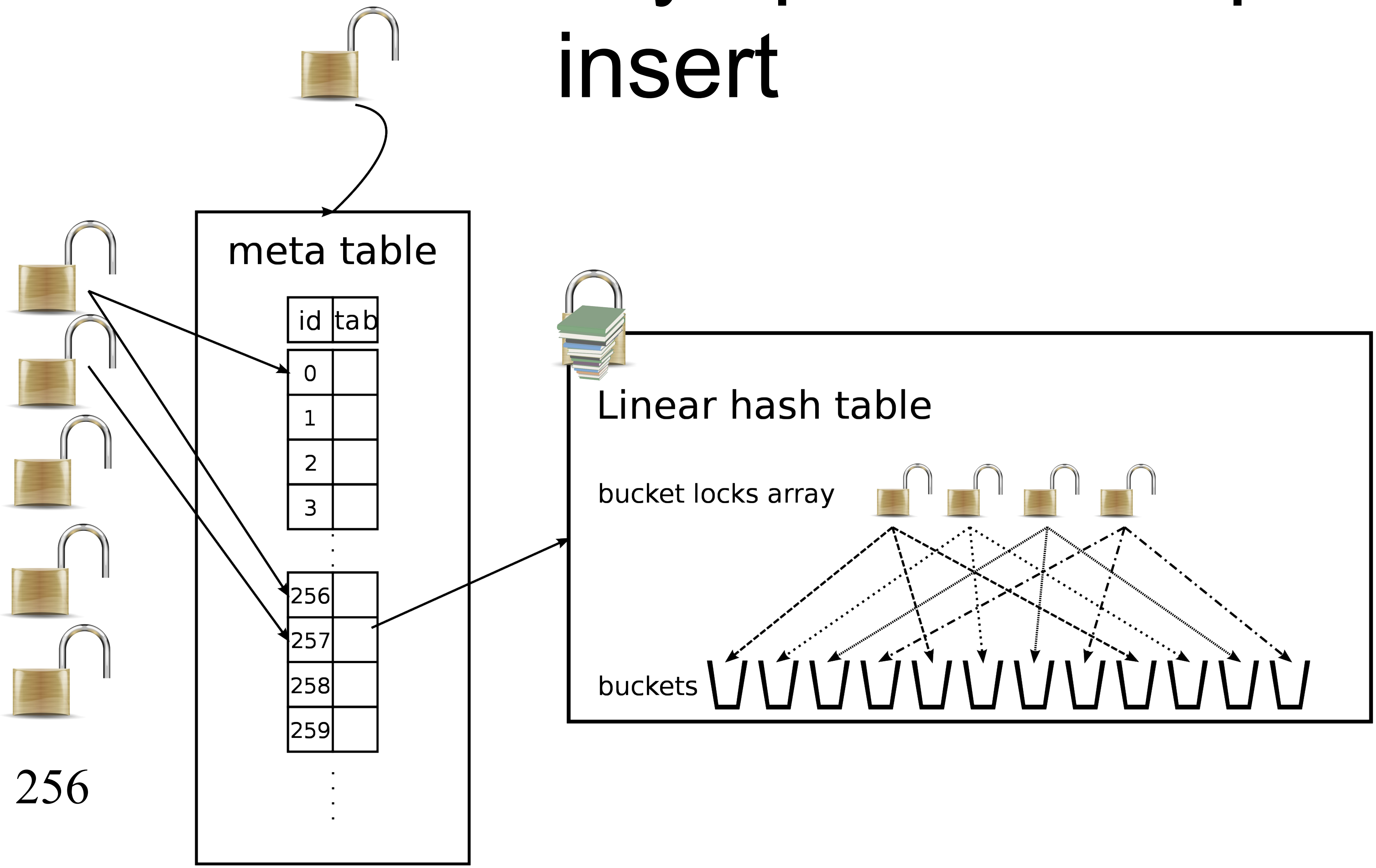
Write concurrency option example insert



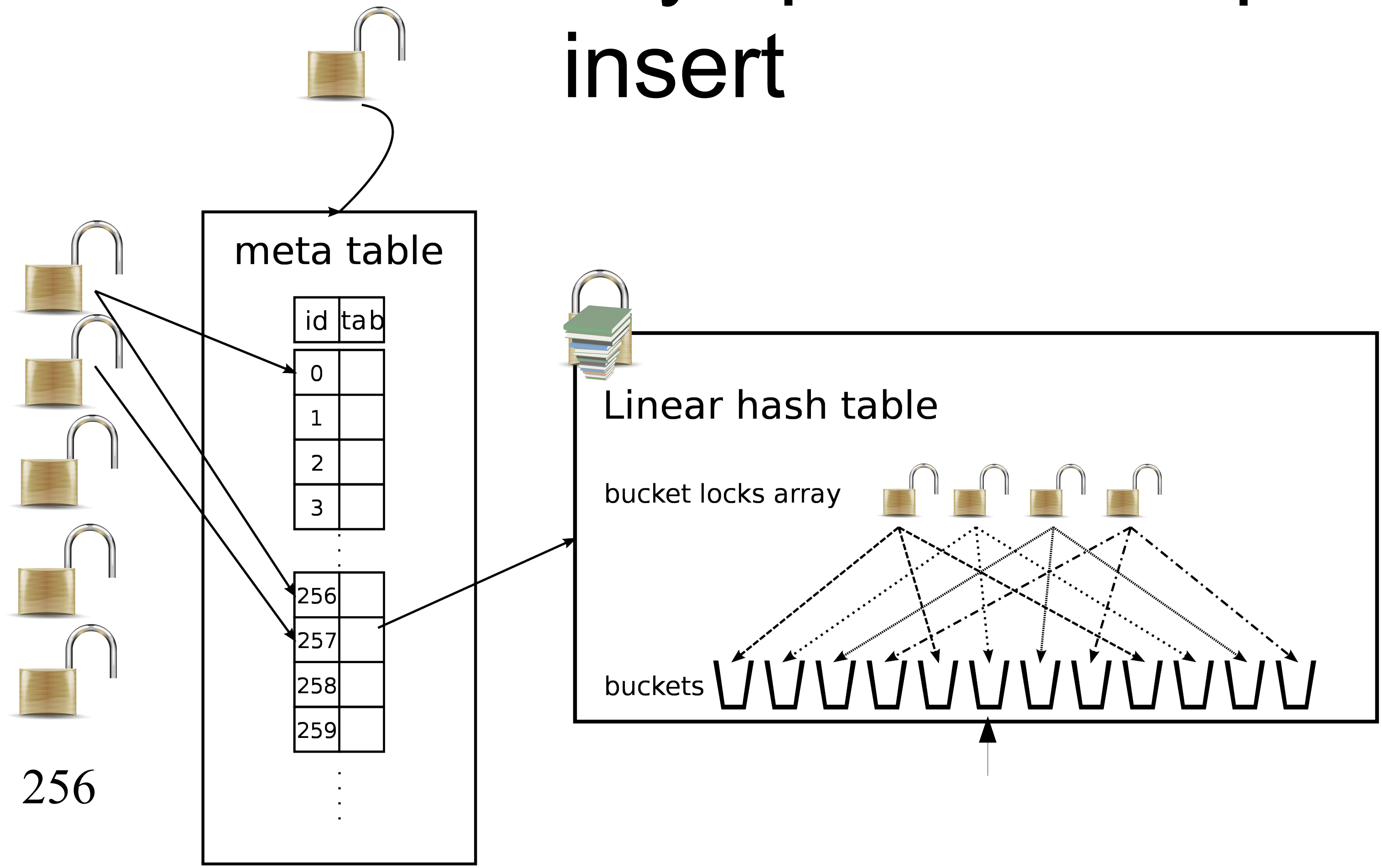
Write concurrency option example insert



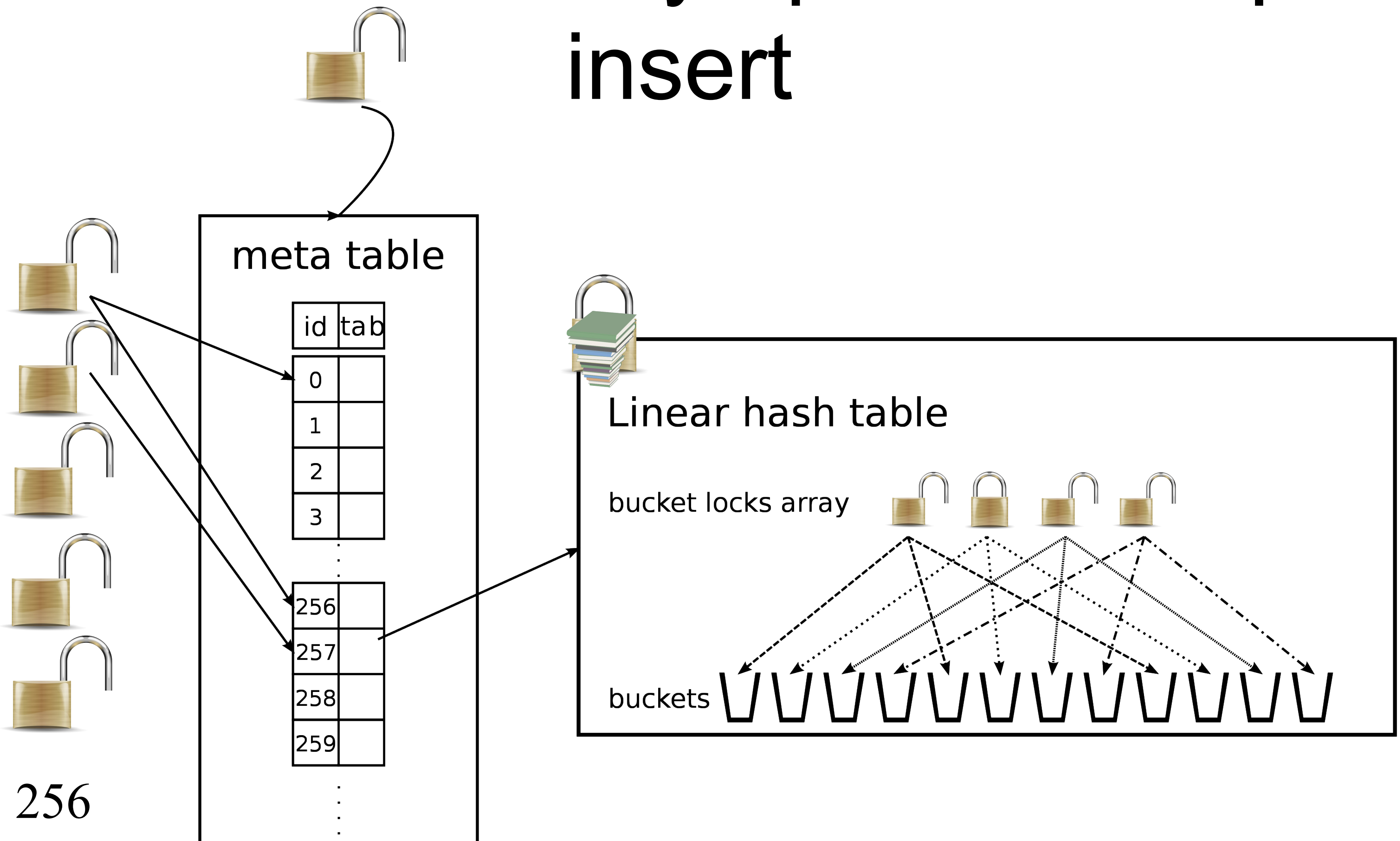
Write concurrency option example insert



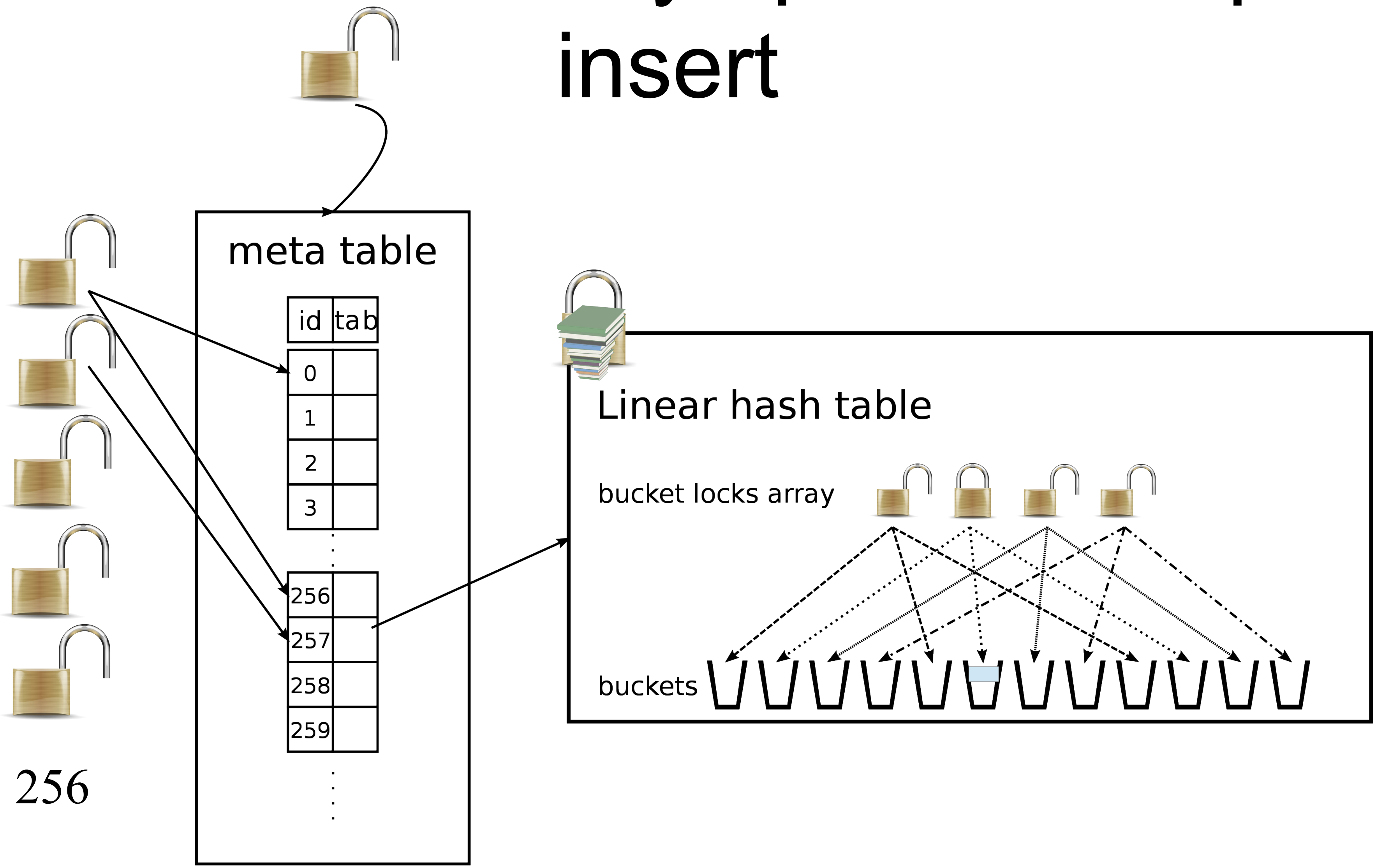
Write concurrency option example insert



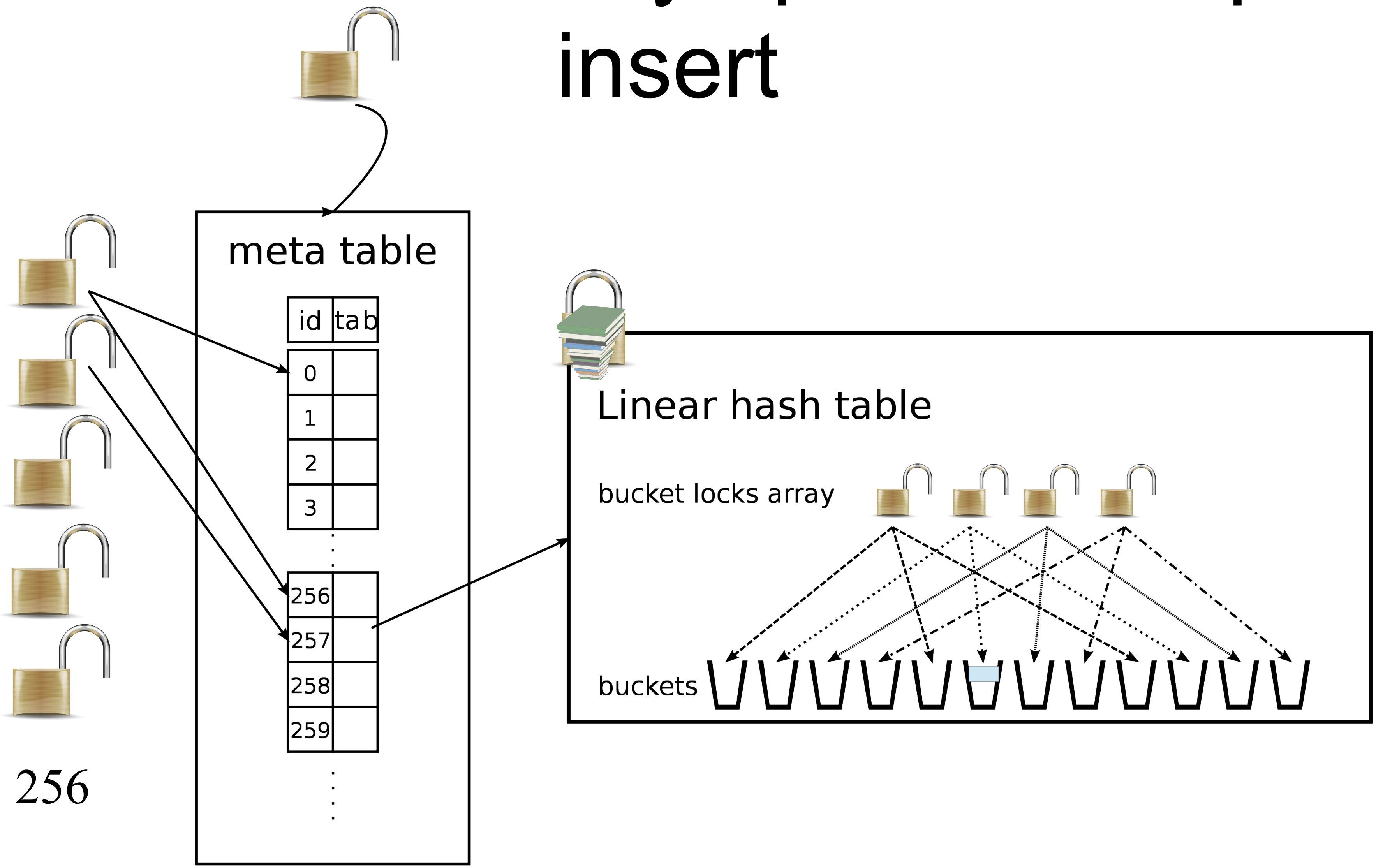
Write concurrency option example insert



Write concurrency option example insert



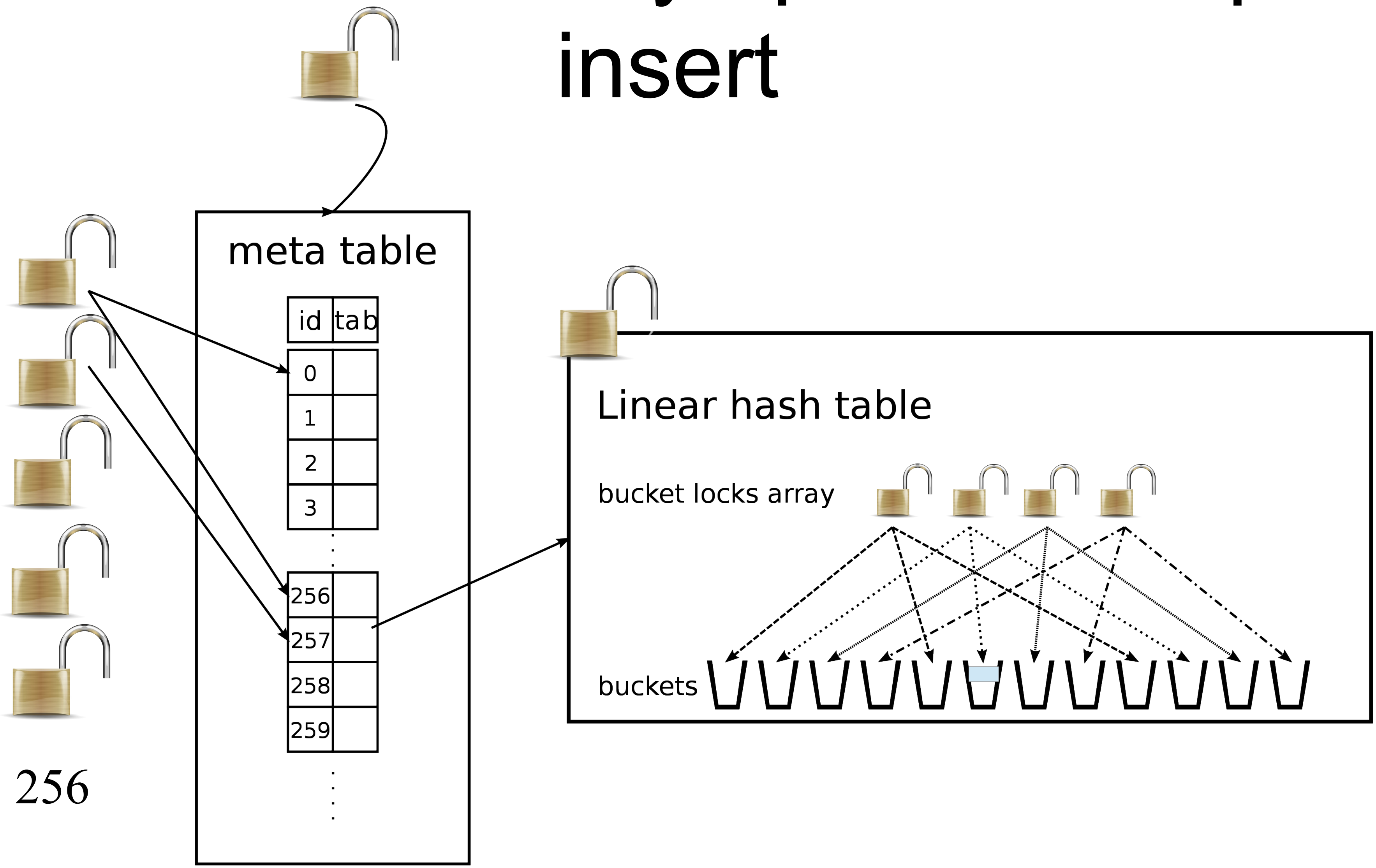
Write concurrency option example insert



meta table

id	tab
0	
1	
2	
3	
256	
257	
258	
259	

Write concurrency option example insert



Reader-Writer Locks

- Writers compete for setting write flag
  - Writers wait for read counter = 0
- Readers increment read counter
  - Wait for write flag
  - After critical section: decrement

Read Concurrency Option

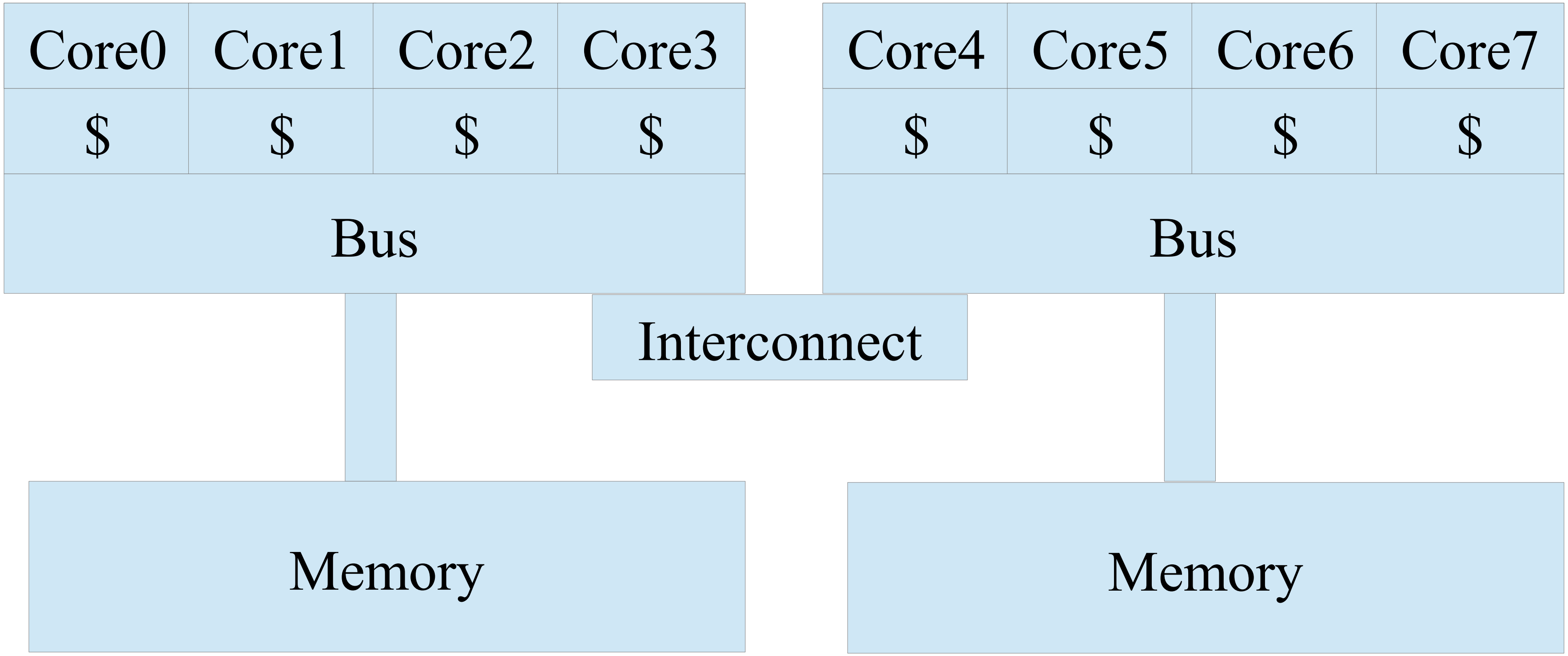
- {read_concurrency, true}
- Introduced in R14B
- Schedulers are mapped to reader groups
- Every reader group has its own read counter

Why Reader Groups?

- We want readers to be fast
- Fast writes = writes in the local cache
- Counters stored in dedicated cache line

Why reader groups?

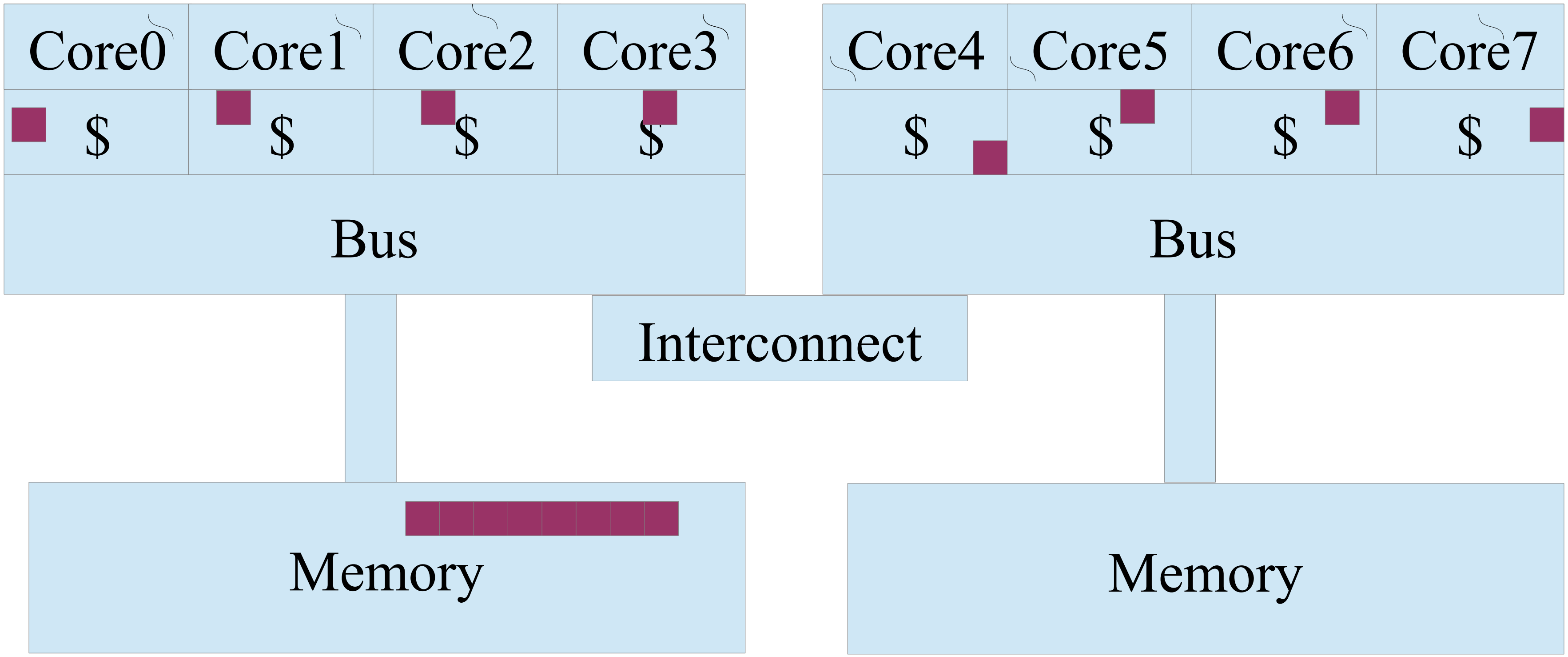
- Modern memory systems are hierarchical
- NUMA (Non-Uniform Memory Access):



Why reader groups?

Reader group counter = ■
Erlang scheduler = ∿

8 schedulers
8 reader groups
(up to 64 in R16B)



Concurrency Options Summary

- write_concurrency
  - (currently) Only on set, bag, duplicate_bag
  - 64 bucket locks
    - Without reader groups
    - read_concurrency → reader groups
  - Reader groups for main table lock
- read_concurrency
  - Works on all tables
  - Reader groups for main table lock

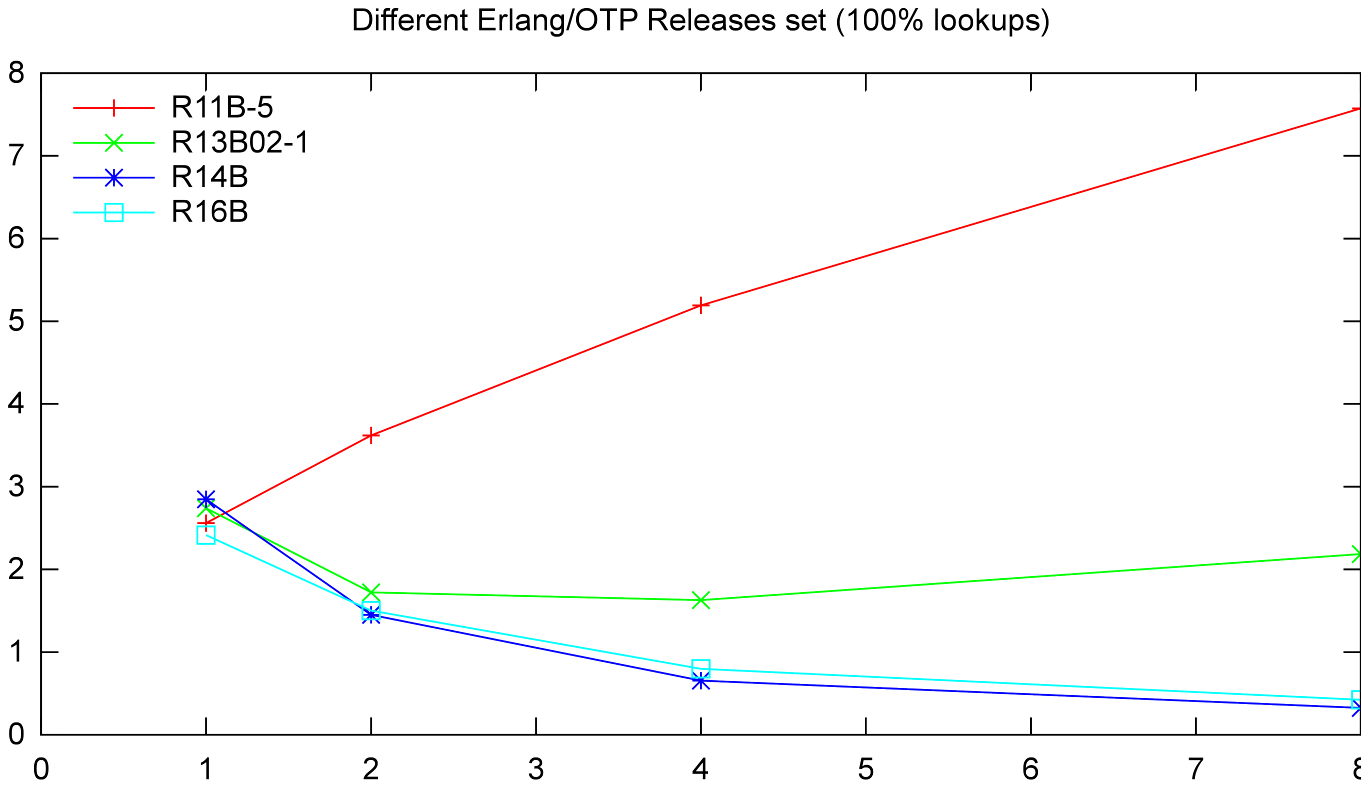
Benchmarks

- Table initialized with ~1 million inserts
- Mixed updates and lookups workload
  - Equal probability for delete and insert
    → size stays approximately the same
- Schedulers bound to cores
- Graphs
  - x-axis: number of schedulers
  - y-axis: time in seconds

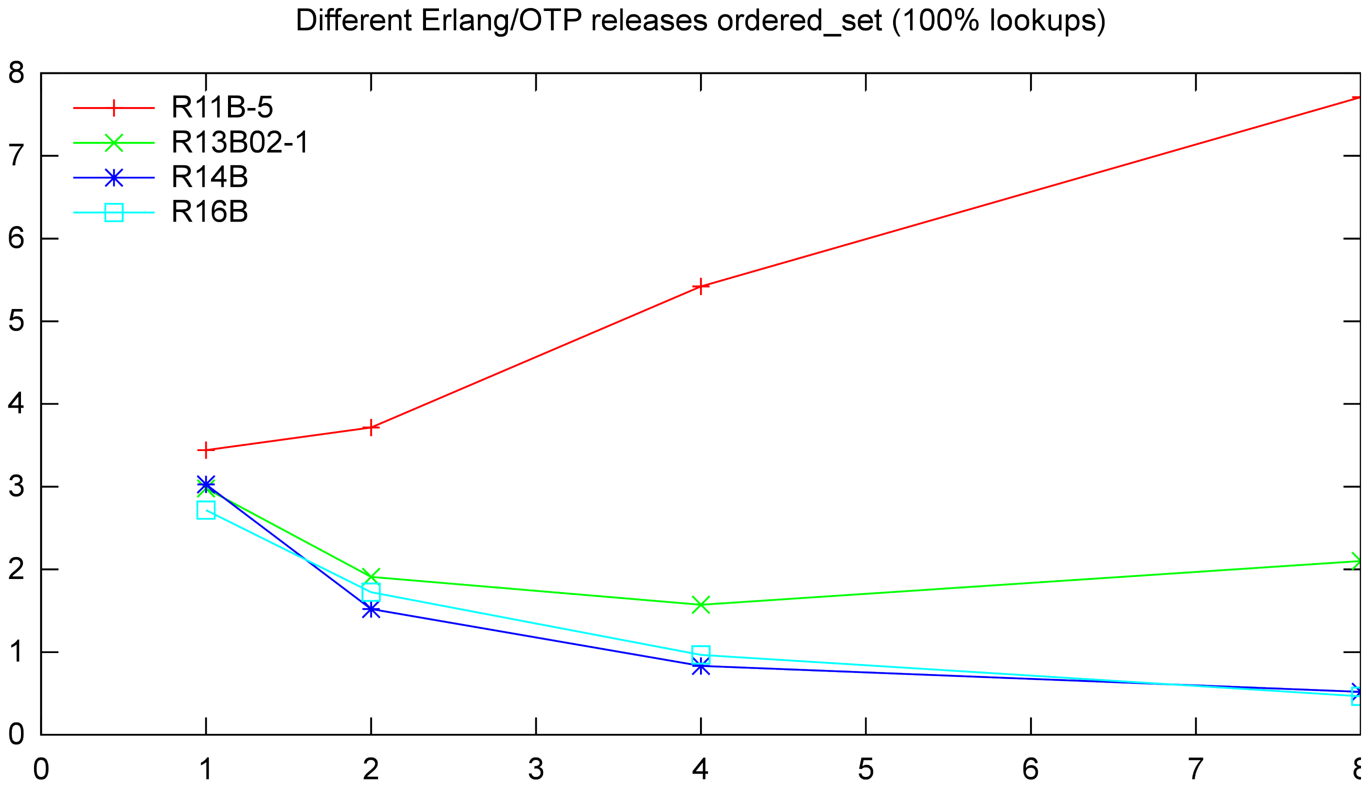
Evolution

- Changes over the years
- R11B → SMP
- R13B02-1 → write_concurrency
- R14B → read_concurrency
- R16B → current

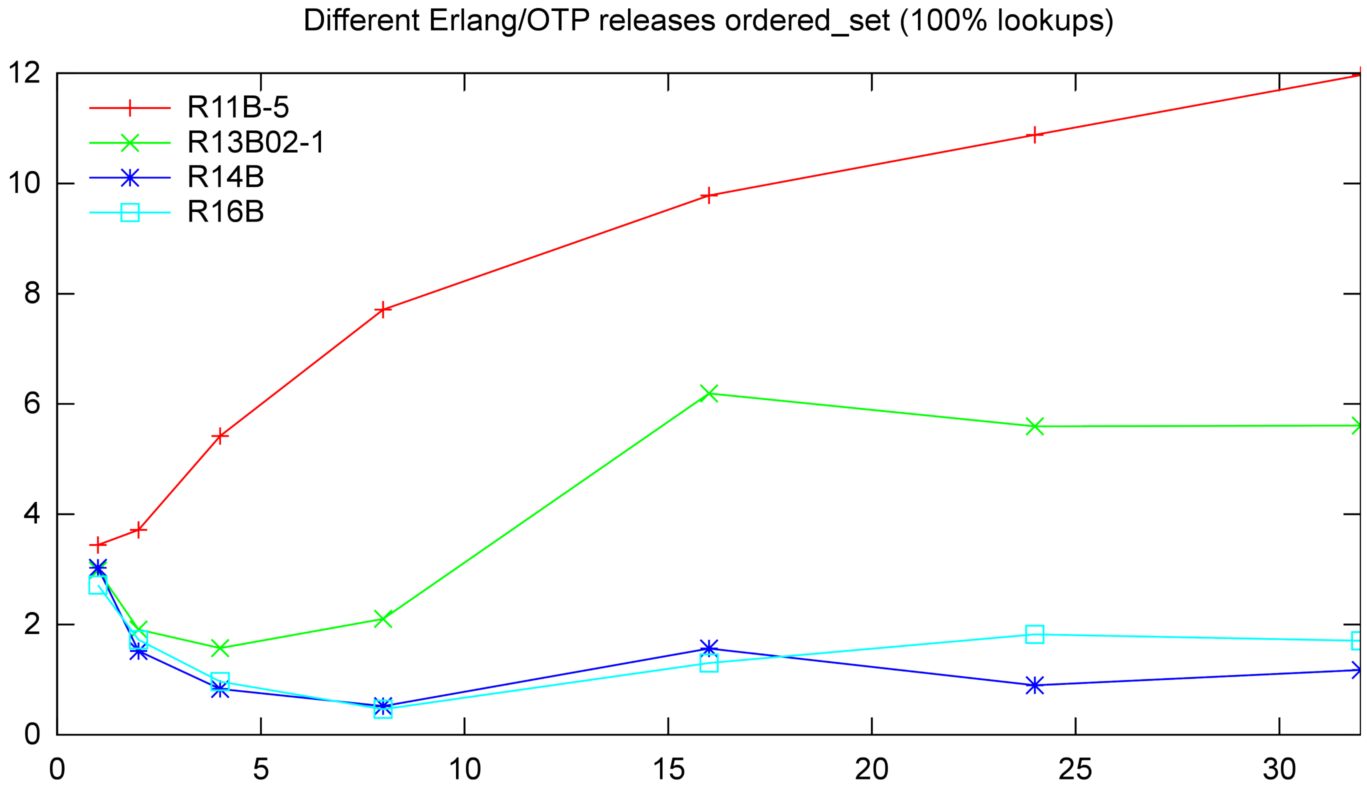
Evolution


Different Erlang/OTP Releases set (100% lookups)
R11B-5
R13B02-1
R14B
R16B
0
1
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8

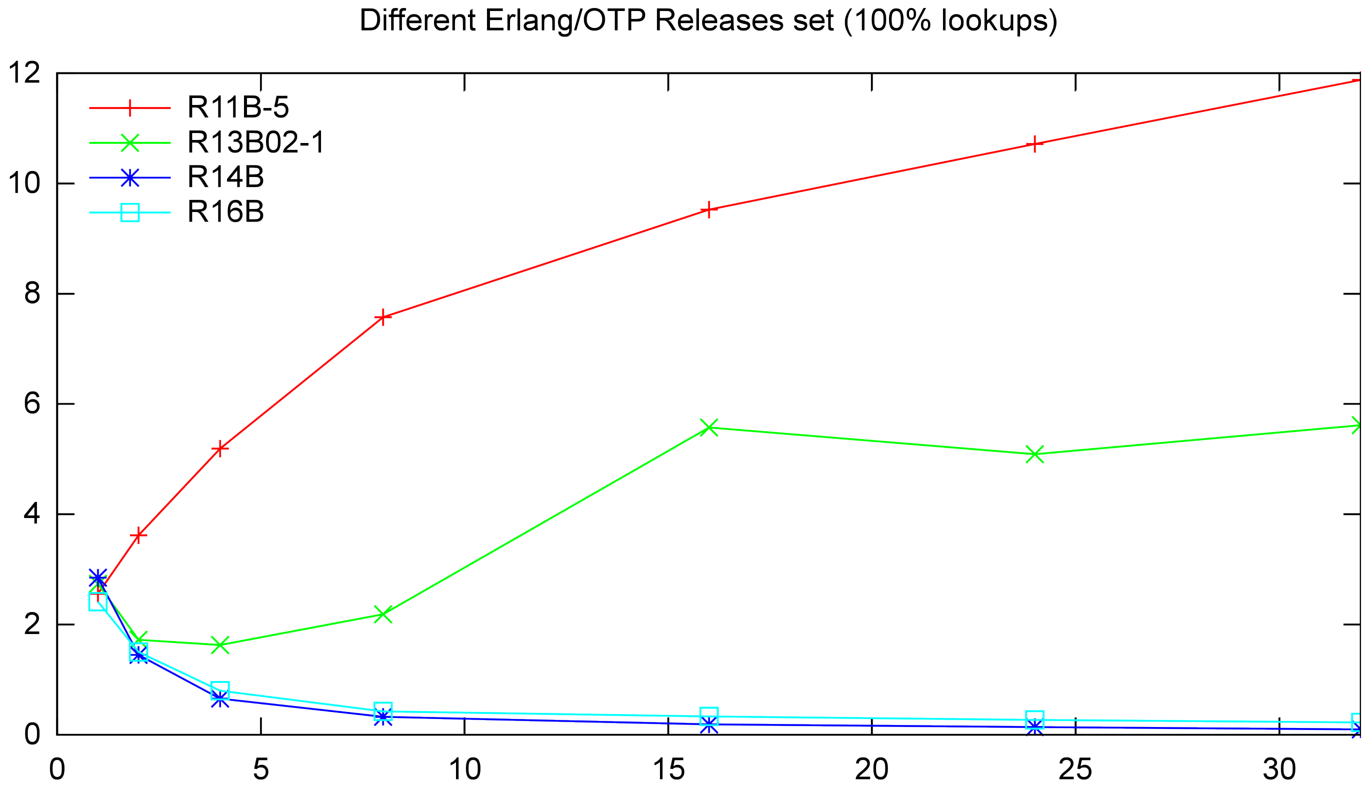
Evolution


Different Erlang/OTP releases ordered_set (100% lookups)
R11B-5
R13B02-1
R14B
R16B
0
1
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8

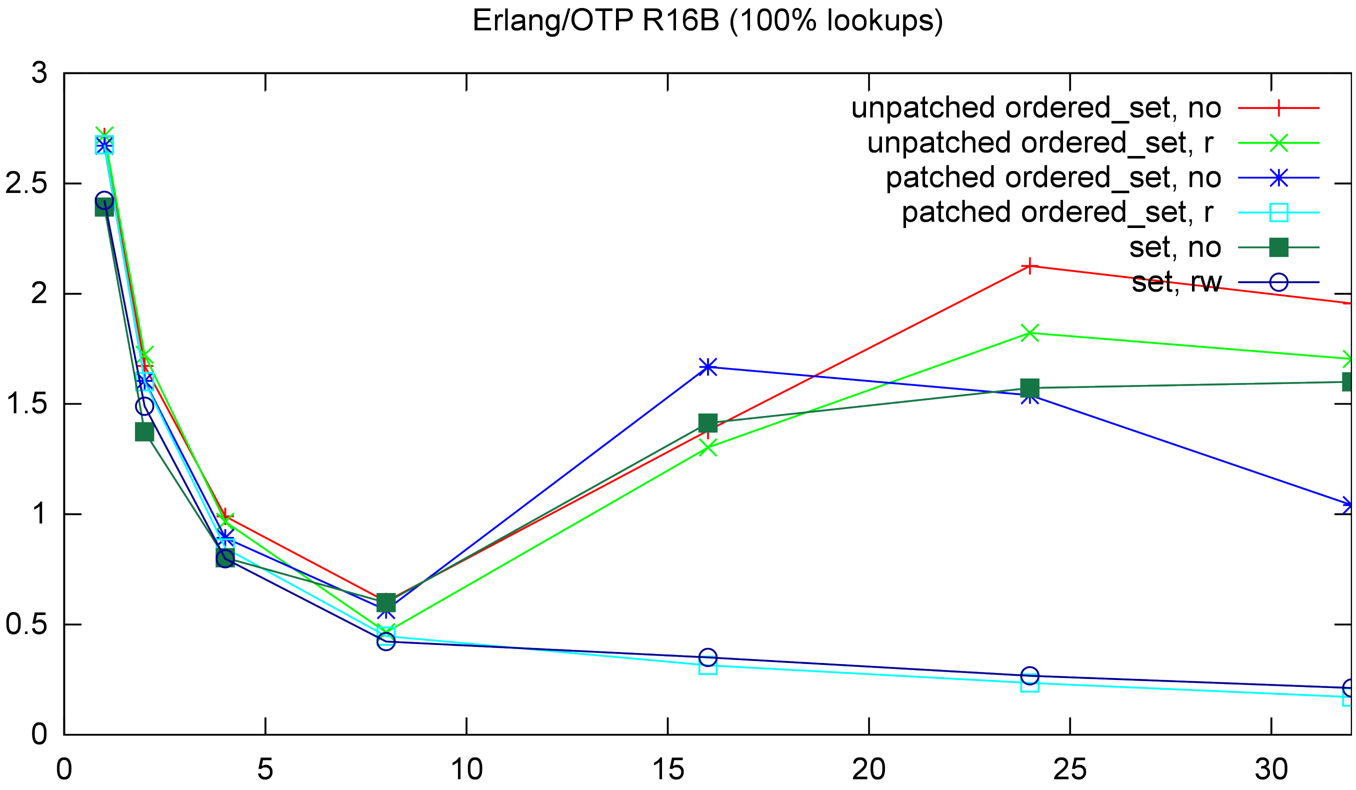
Evolution


Different Erlang/OTP releases ordered_set (100% lookups)
R11B-5
R13B02-1
R14B
R16B
0
2
4
6
8
10
12
0
5
10
15
20
25
30

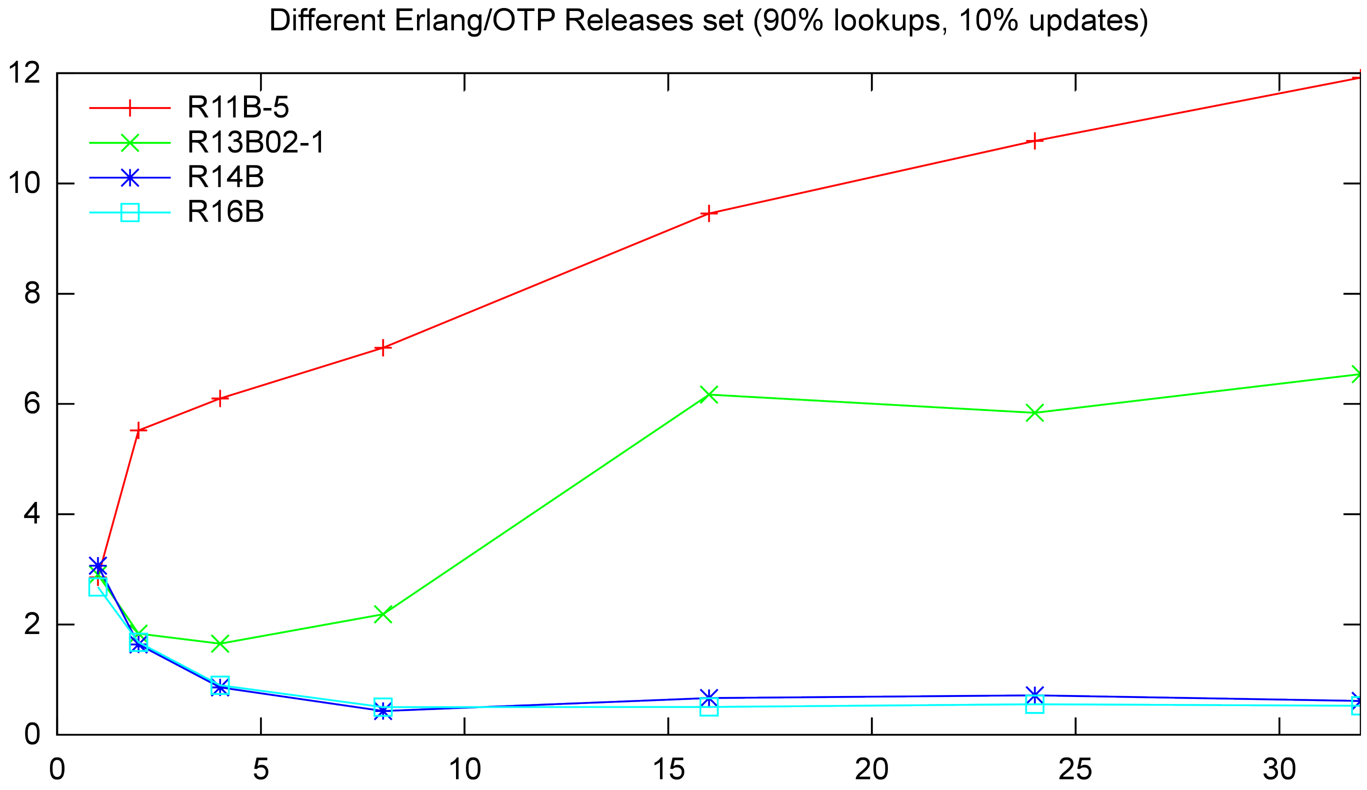
Evolution


Different Erlang/OTP Releases set (100% lookups)
R11B-5
R13B02-1
R14B
R16B
0
2
4
6
8
10
12
0
5
10
15
20
25
30

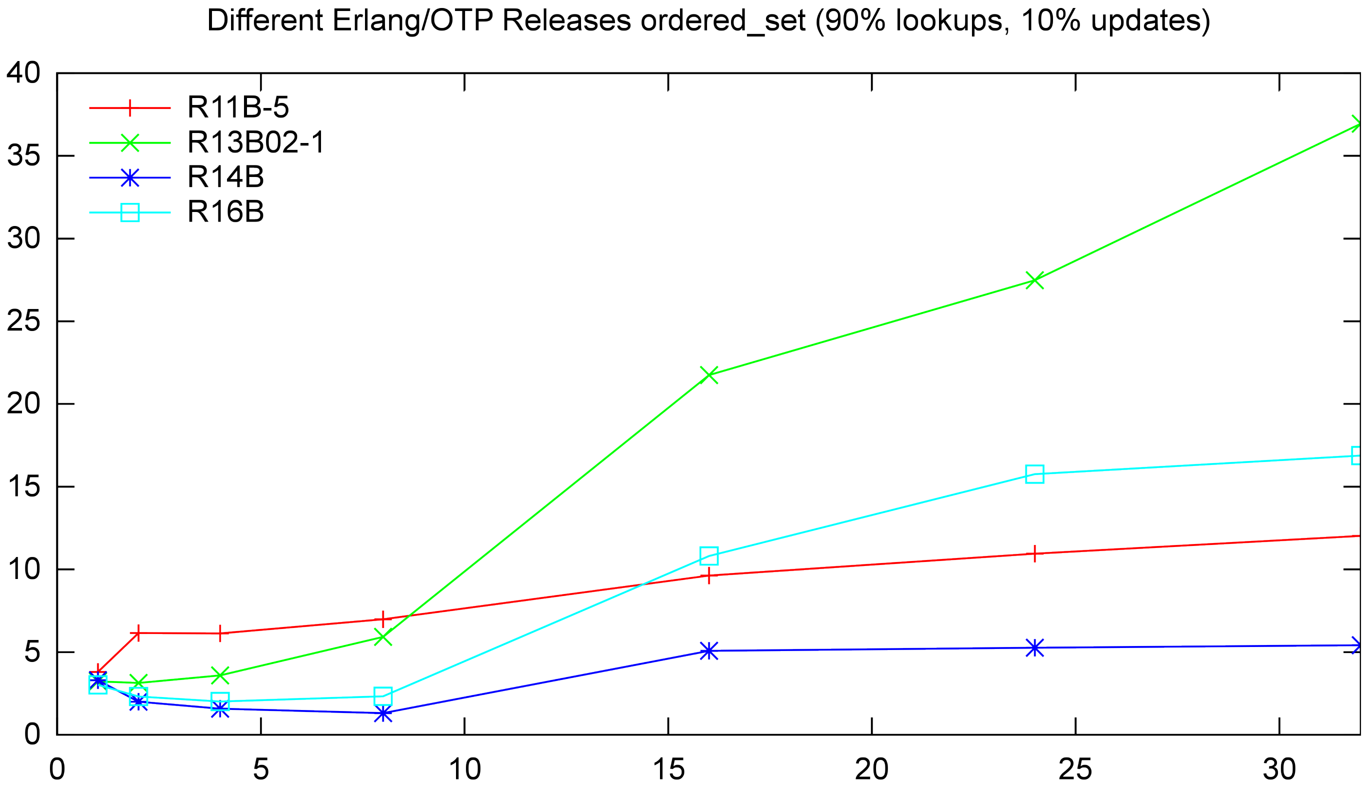
Next Release?



Evolution


Different Erlang/OTP Releases set (90% lookups, 10% updates)
R11B-5
R13B02-1
R14B
R16B
0
2
4
6
8
10
12
0
5
10
15
20
25
30

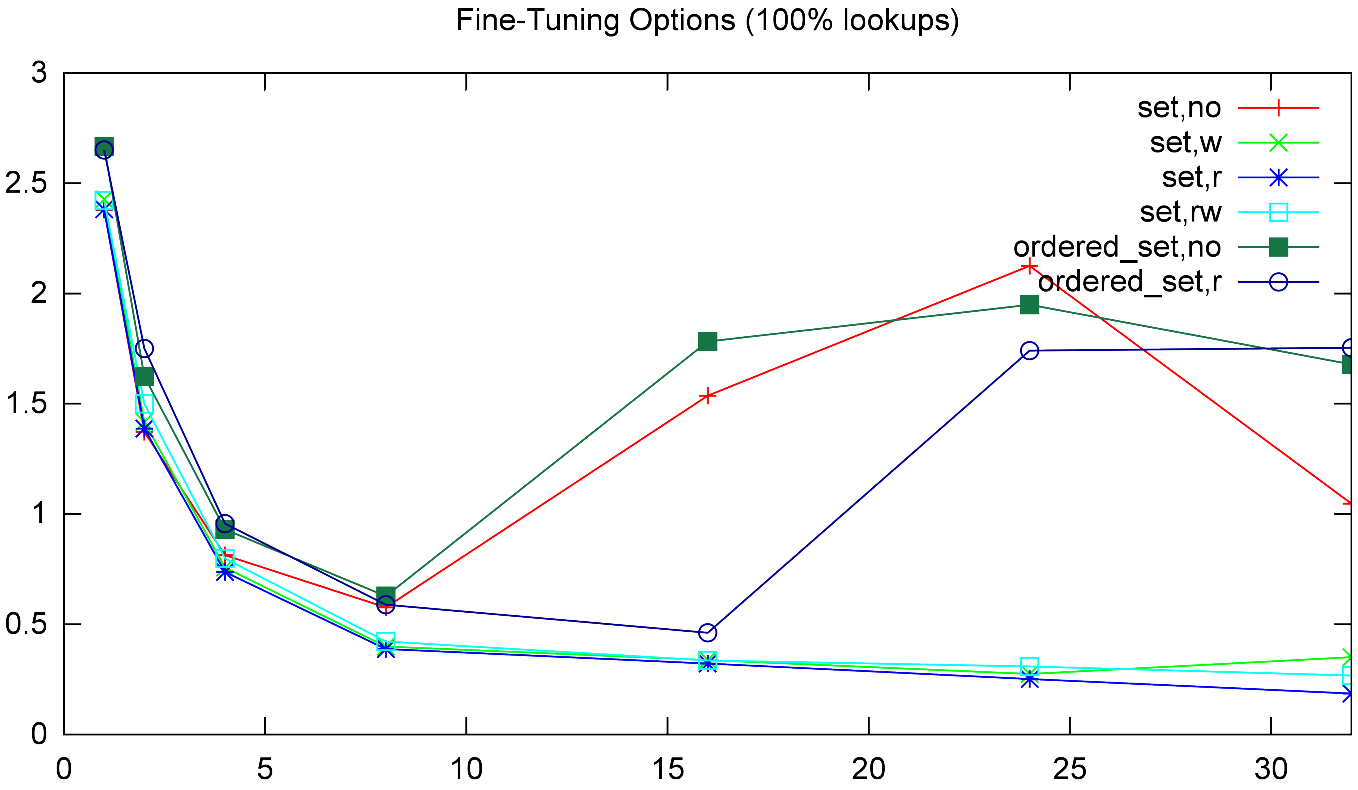
Evolution


Different Erlang/OTP Releases ordered_set (90% lookups, 10% updates)
R11B-5
R13B02-1
R14B
R16B
0
5
10
15
20
25
30
35
40
0
5
10
15
20
25
30

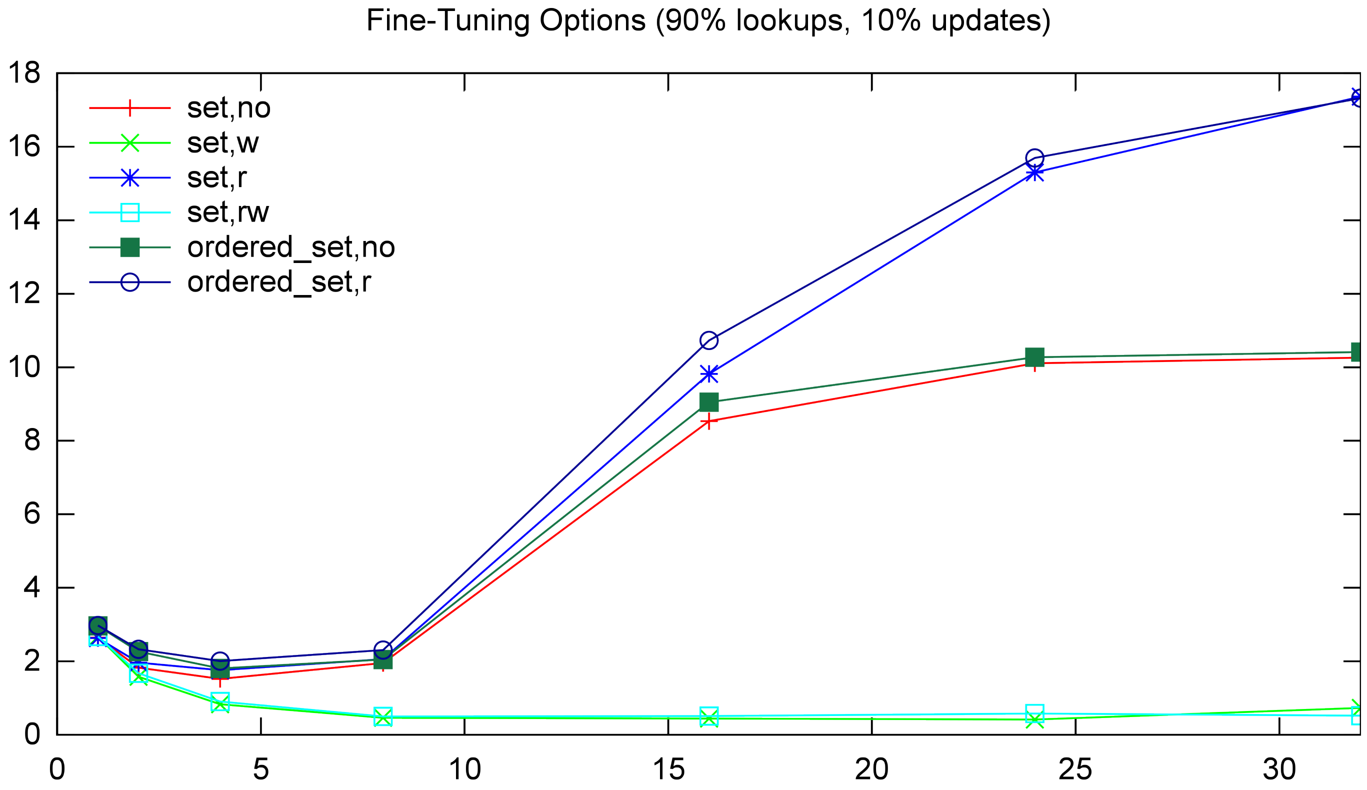
R16B concurrency options

- read_concurrency
- write_concurrency
- or both

R16B concurrency options



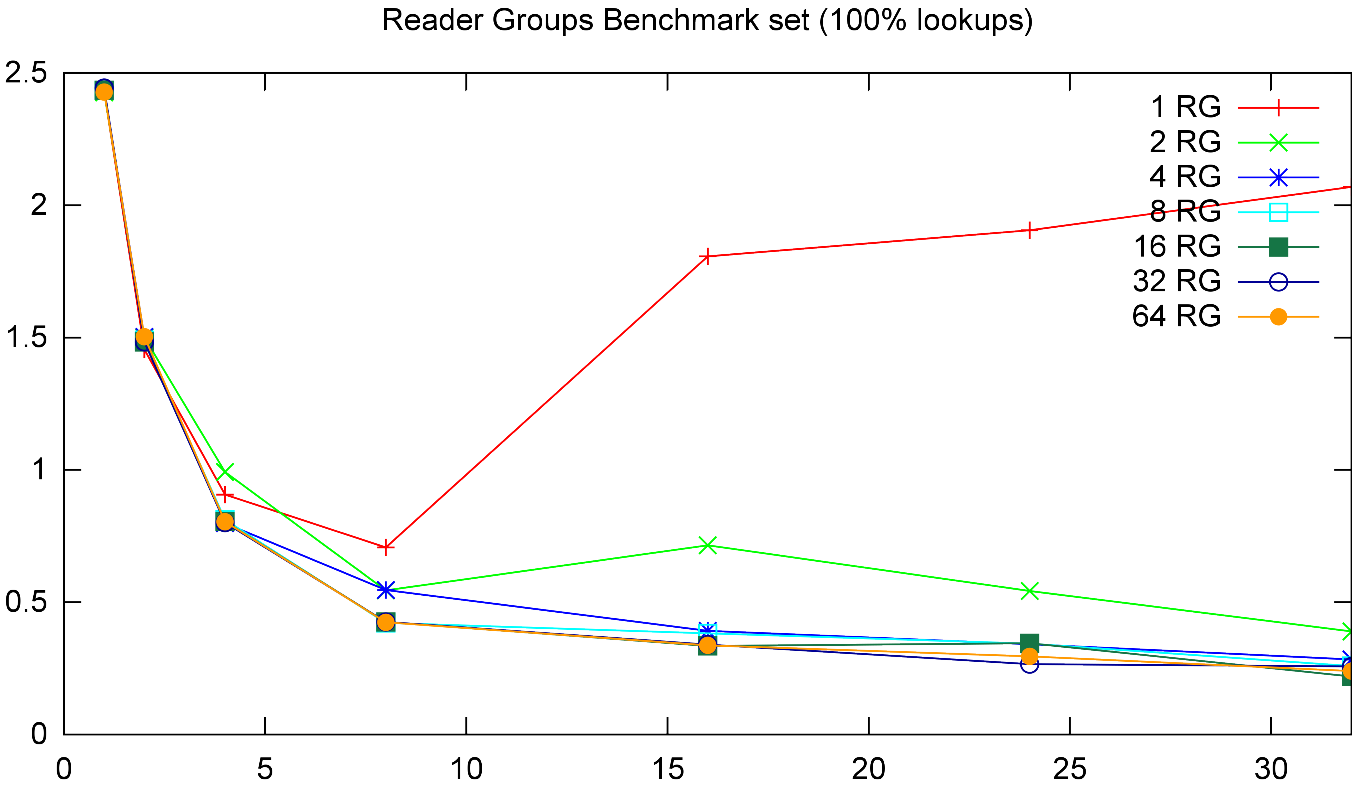
R16B concurrency options



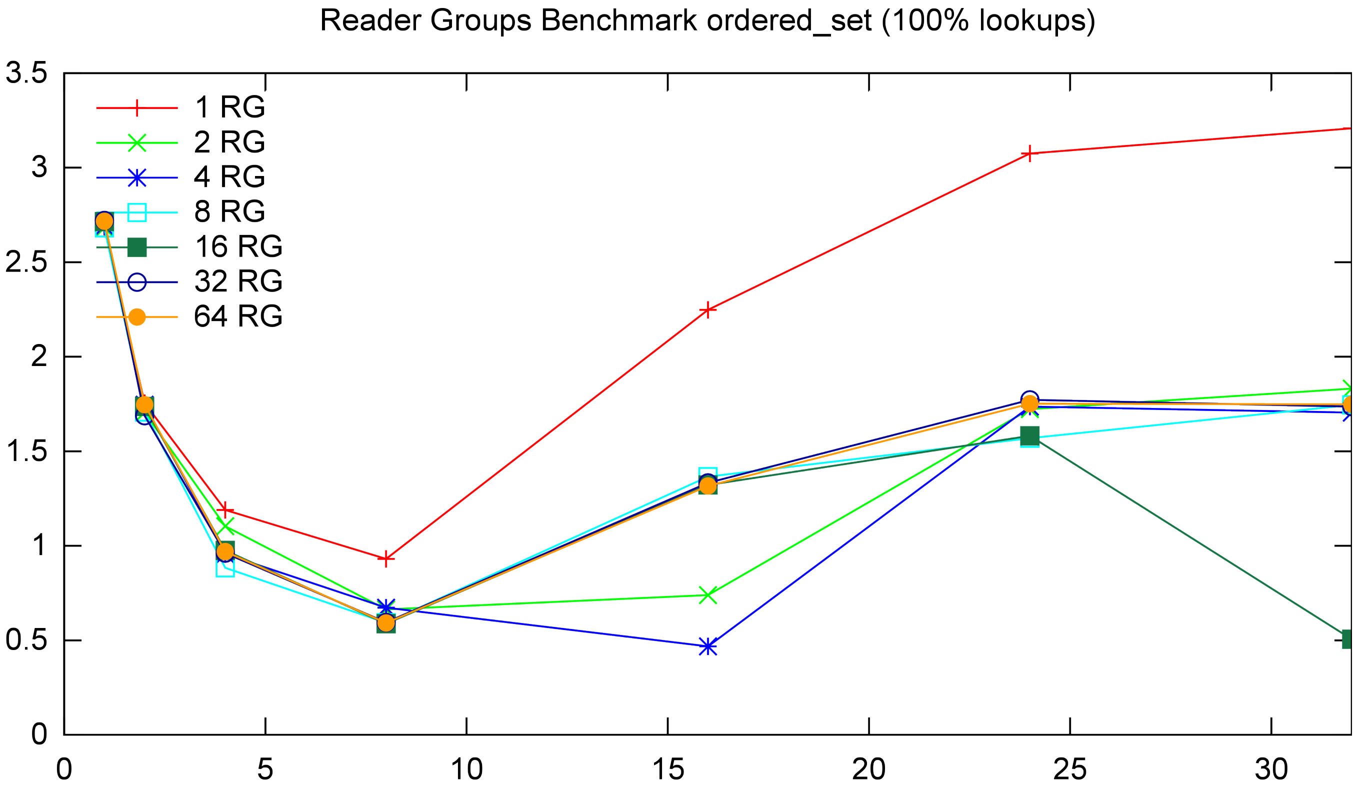
Reader Groups

- +rg command line parameter
- Default: 64
- Maximum number of reader groups
  - real: max(rg, #schedulers)
- Optimization for reads

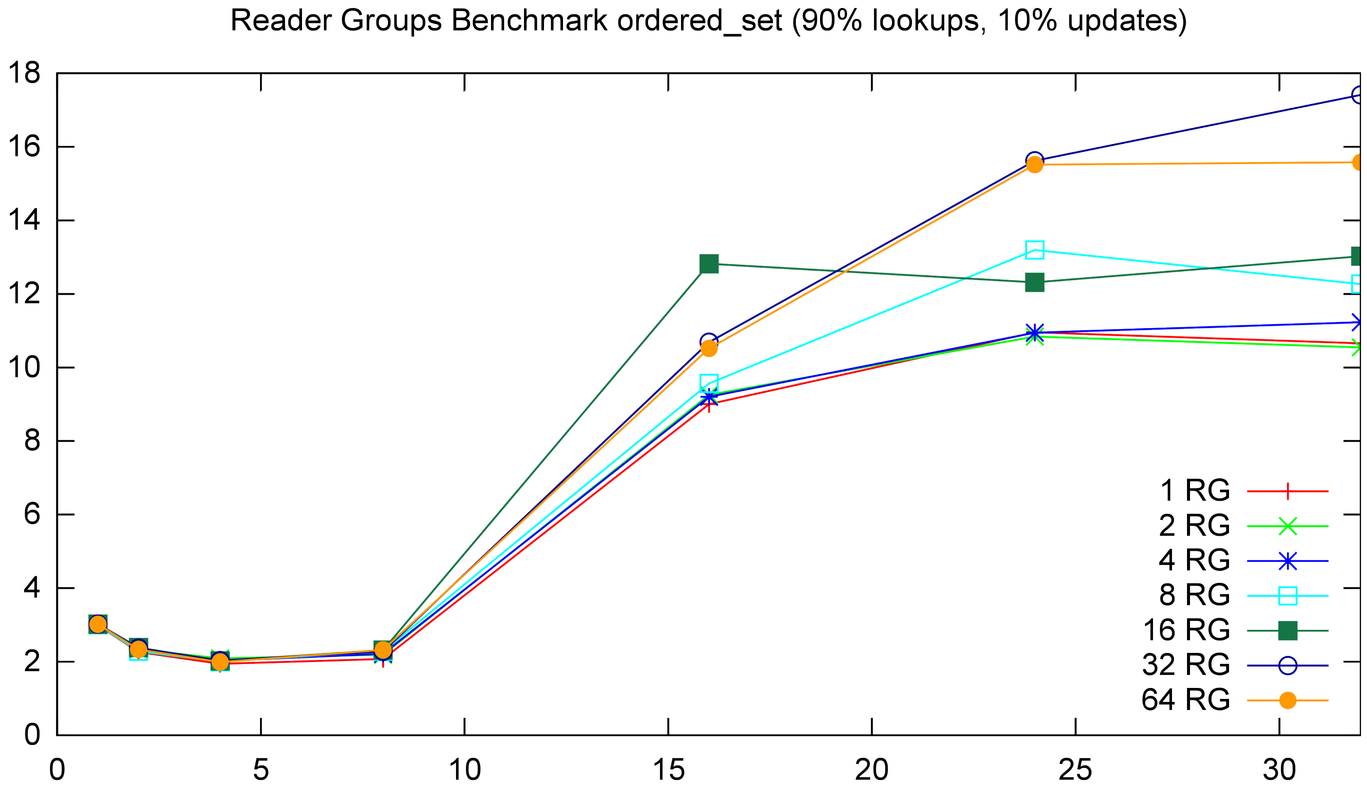
Reader Groups (+rg option)



Reader Groups (+rg option)



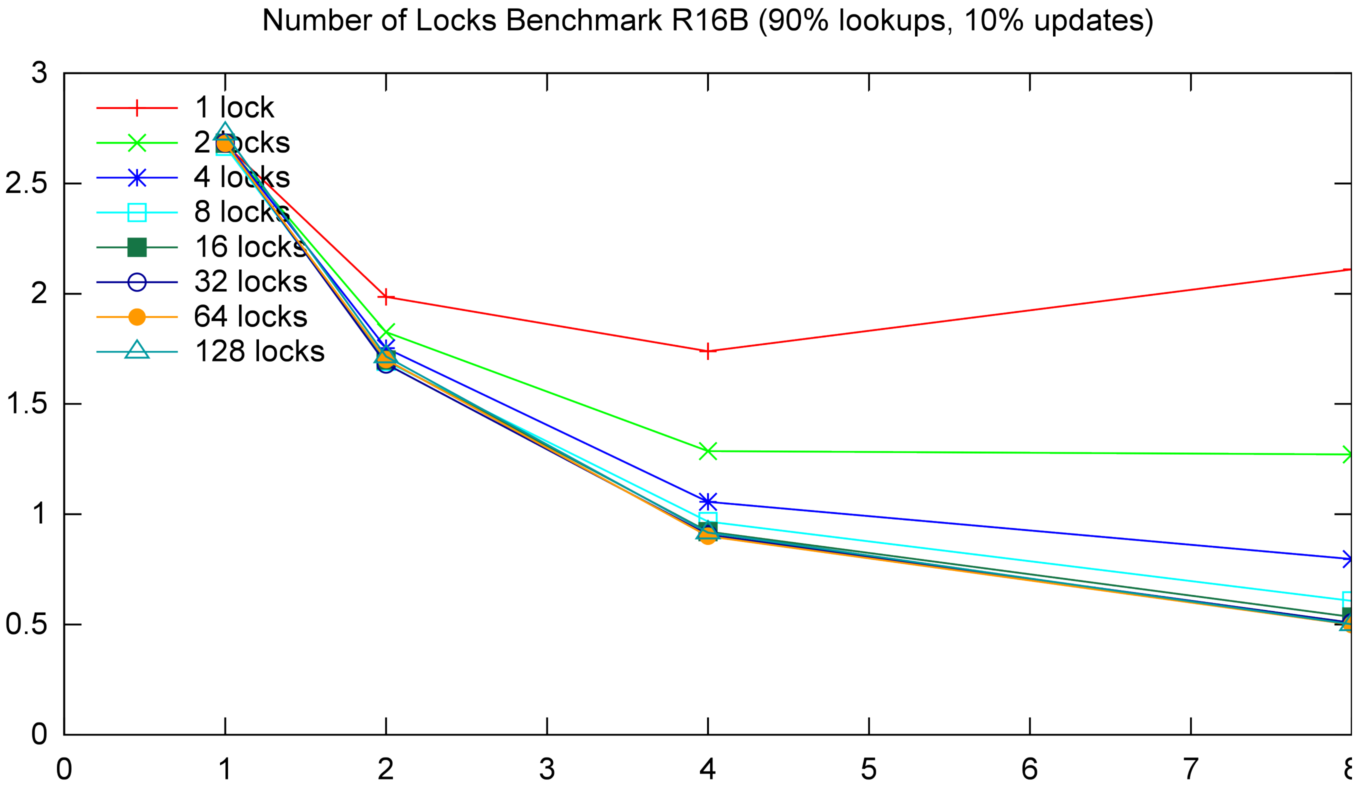
Reader Groups (+rg option)



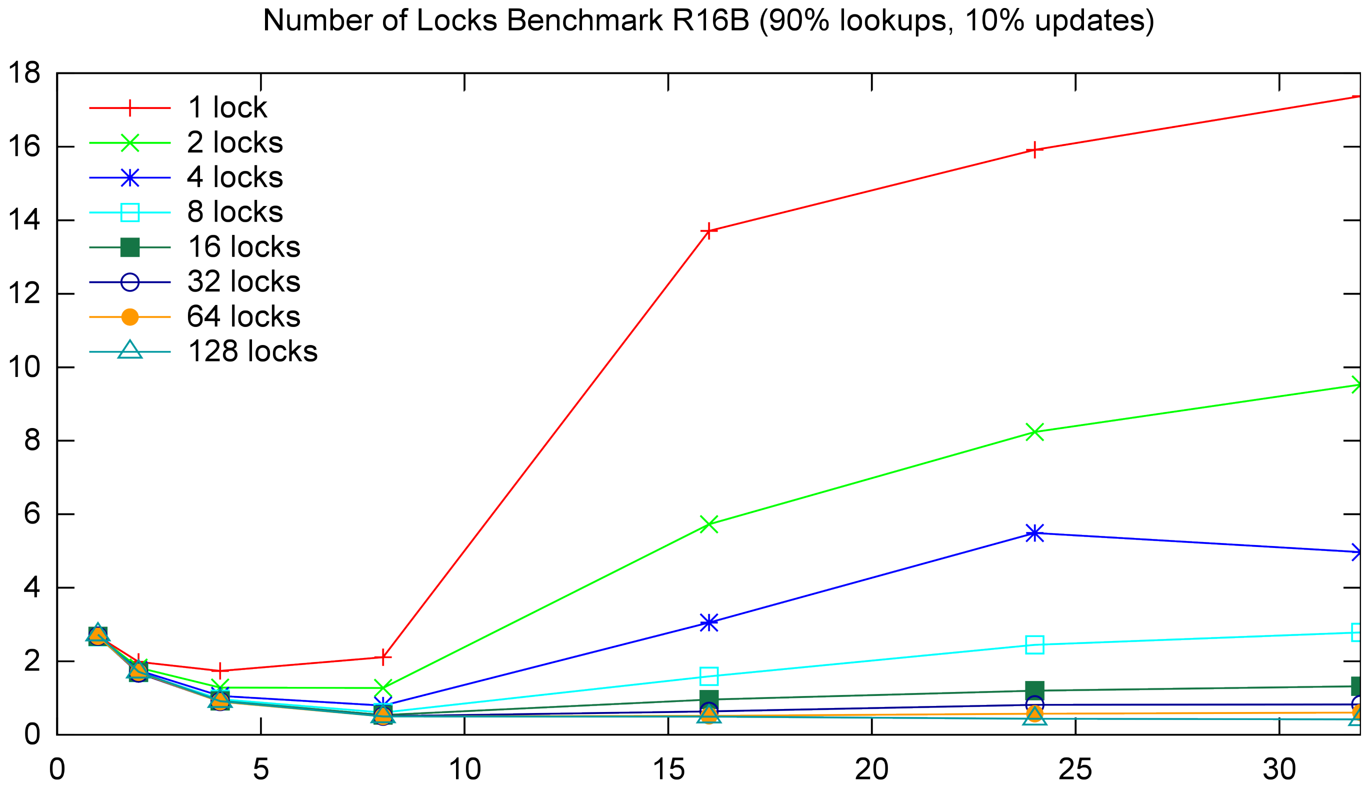
Number of Bucketlocks

- Compiled in
- R16B: 16 → 64
- Important?

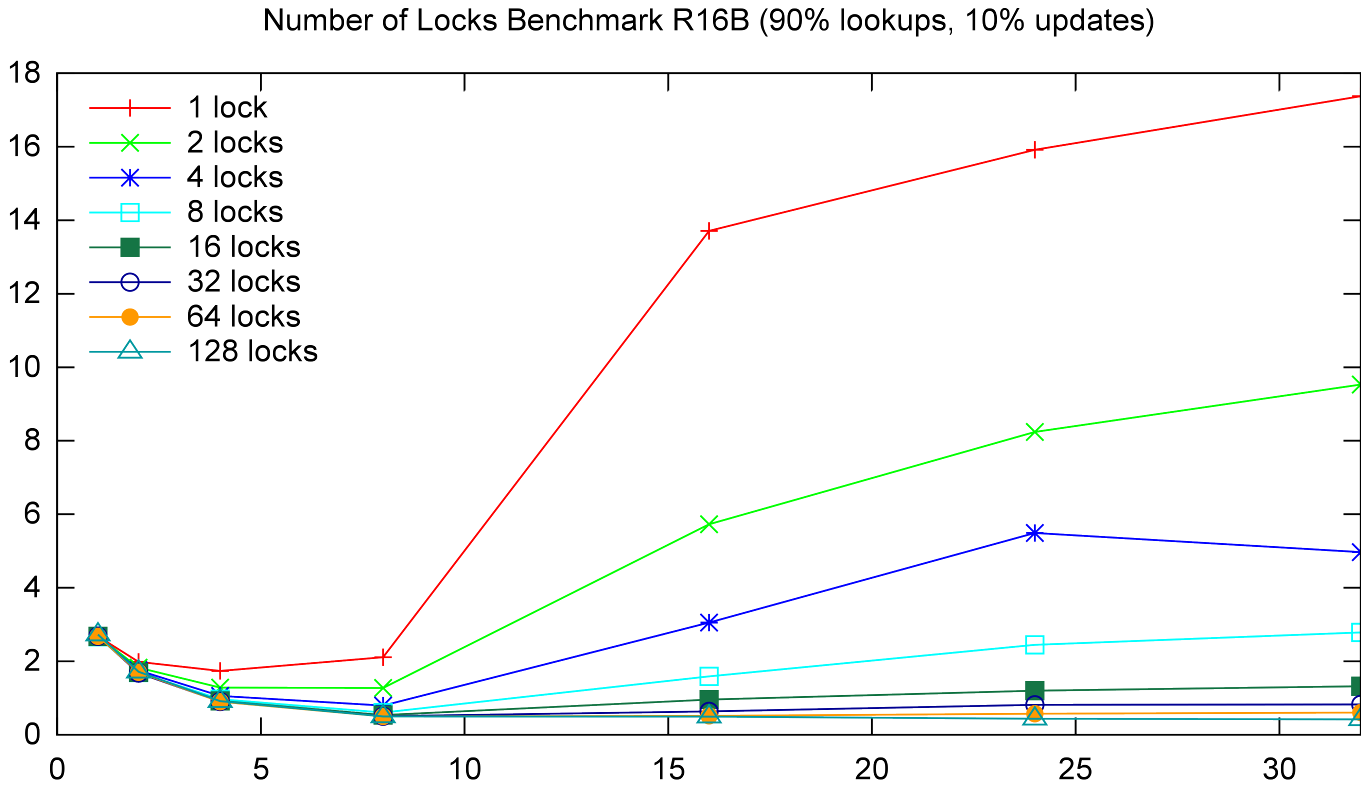
Number of Bucketlocks



Number of Bucketlocks



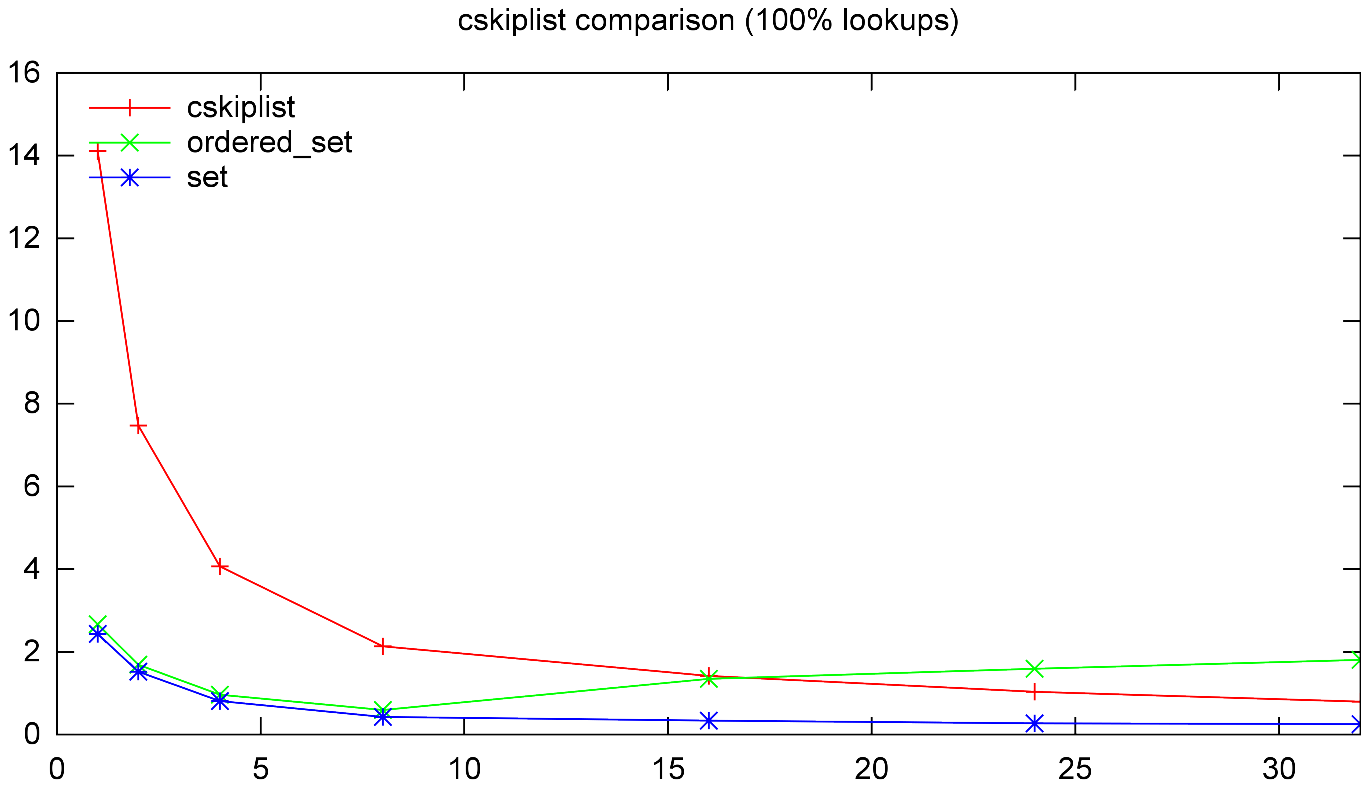
Number of Bucketlocks



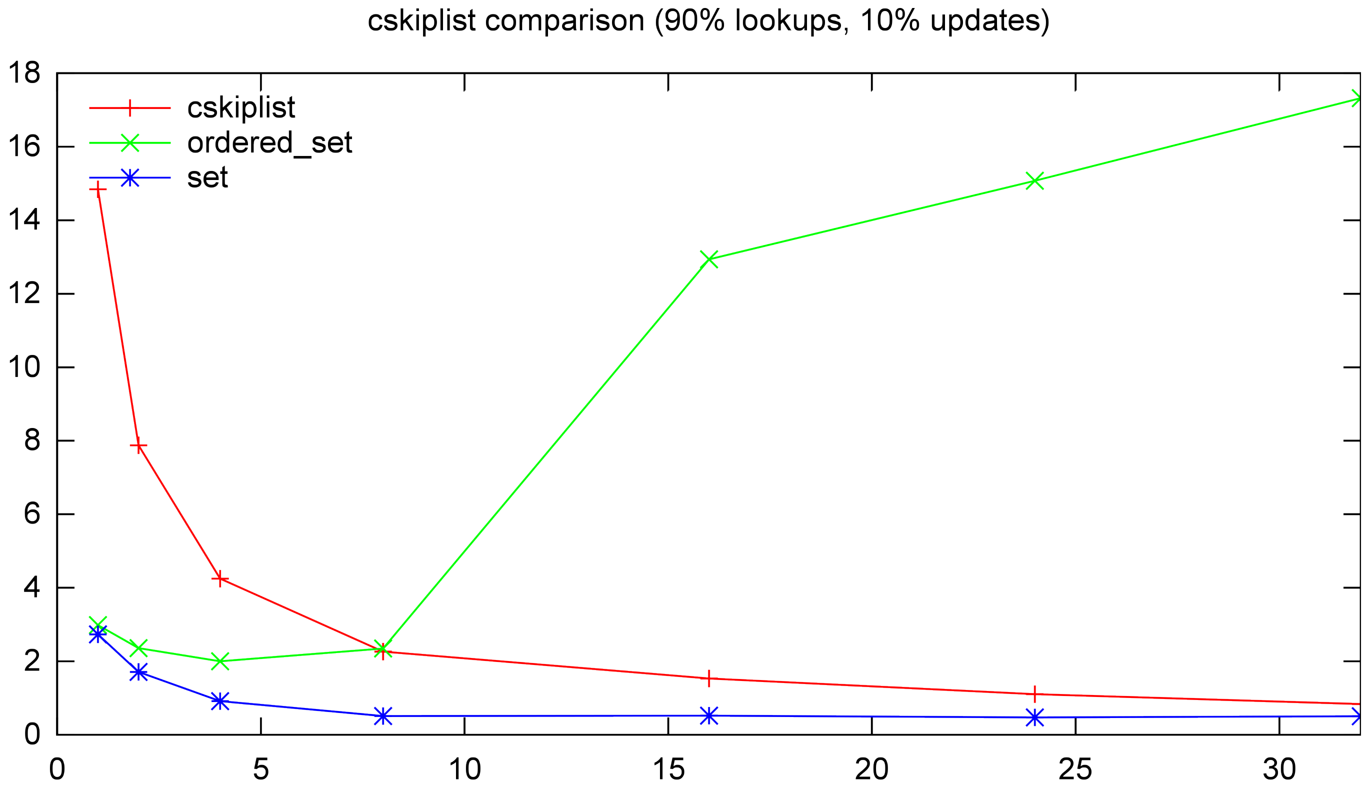
Other Backends?

- New datastructures required?
- Example: Concurrent skiplist
  - Experimental implementation
  - Not lockfree
  - Not optimized

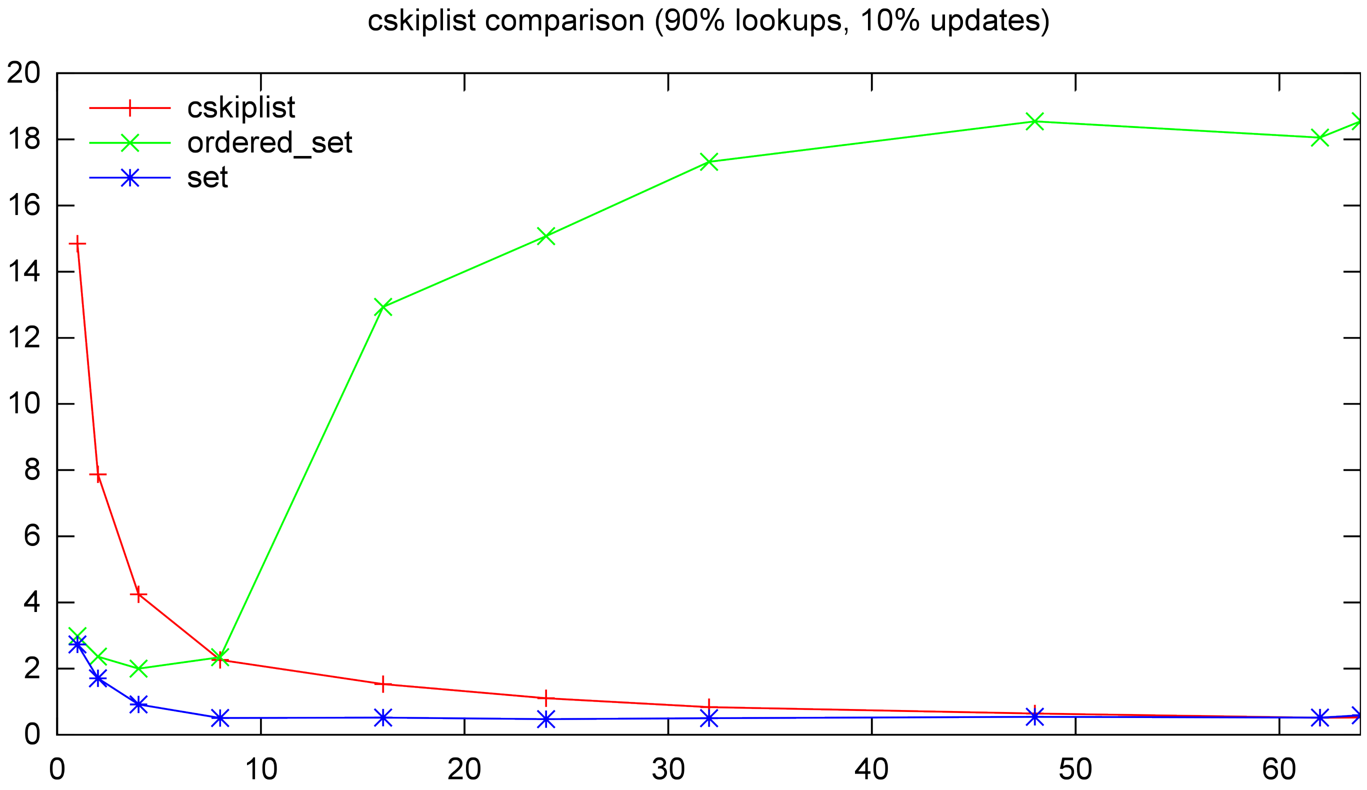
Concurrent Skiplist Backend



Concurrent Skiplist Backend



Concurrent Skiplist Backend



Scaling ETS

- Does it exist?
- ordered_set needs to be fixed or replaced
- Locking is (still) a problem, but got better
- NUMA is a problem
- Reader groups may be not that important

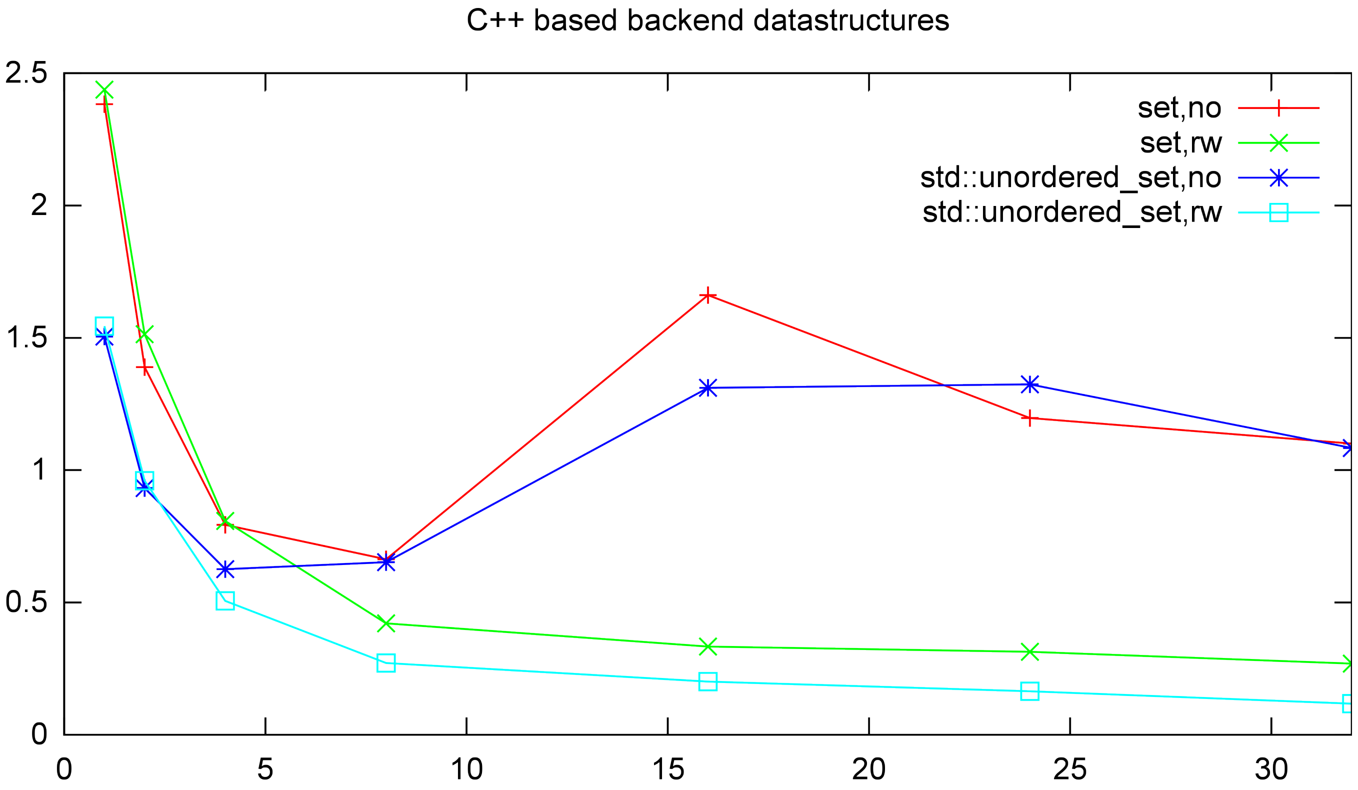
Lessons learned

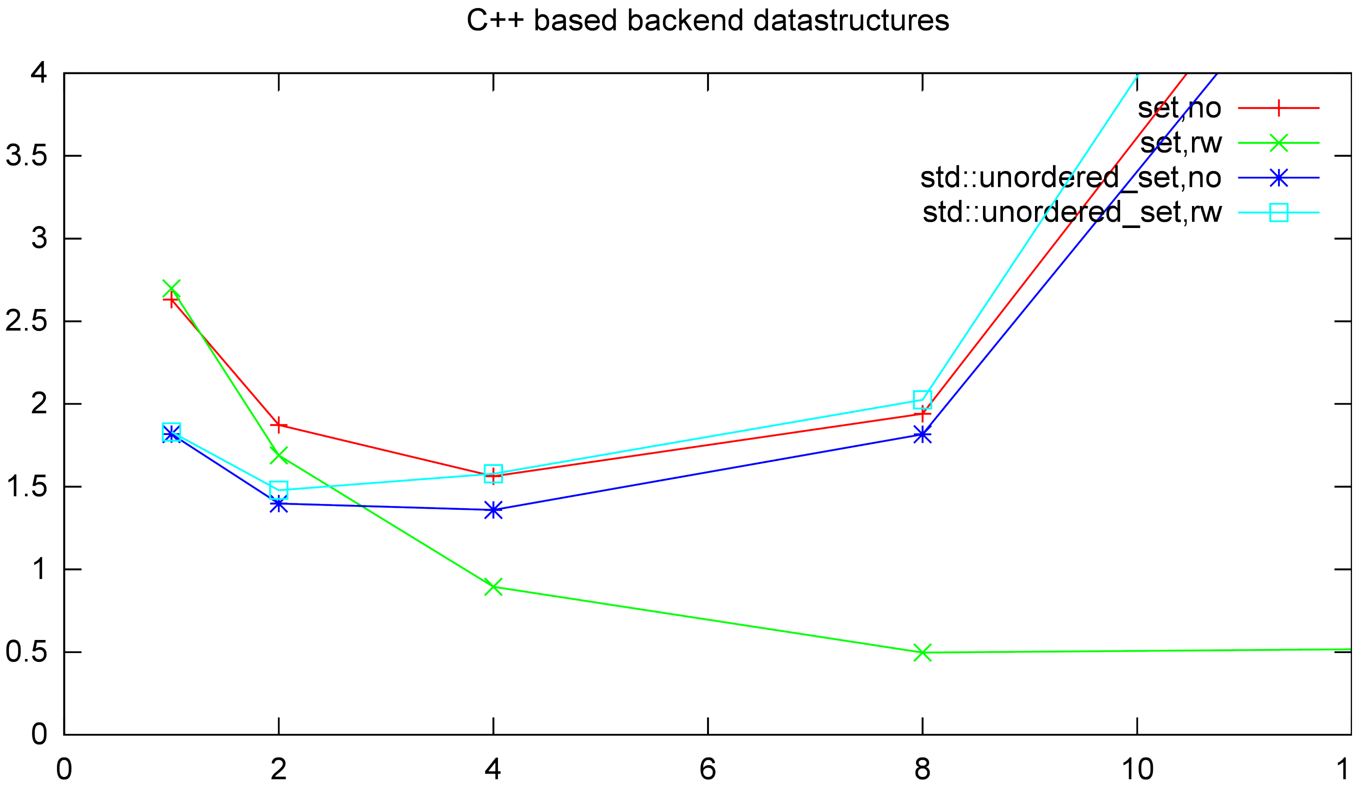
- Use pinning on NUMA
  - Memory communication cost
- Use read_concurrency when doing only lookups
- Use write_concurrency
  - set, bag, duplicate_bag
- Measure your use case when combining them

Conclusions

- Increasing number of bucket locks helps
- New datastructure backends can help
- Too many locks
  - Meta table
  - Table
  - Buckets
- A scaling ordered_set could outperform set

Questions?


C++ based backend datastructures
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
set,no
set,rw
std::unordered_set,no
std::unordered_set,rw


C++ based backend datastructures
set,no
set,rw
std::unordered_set,no
std::unordered_set,rw
4
3.5
3
2.5
2
1.5
1
0.5
0
0
2
4
6
8
10
1