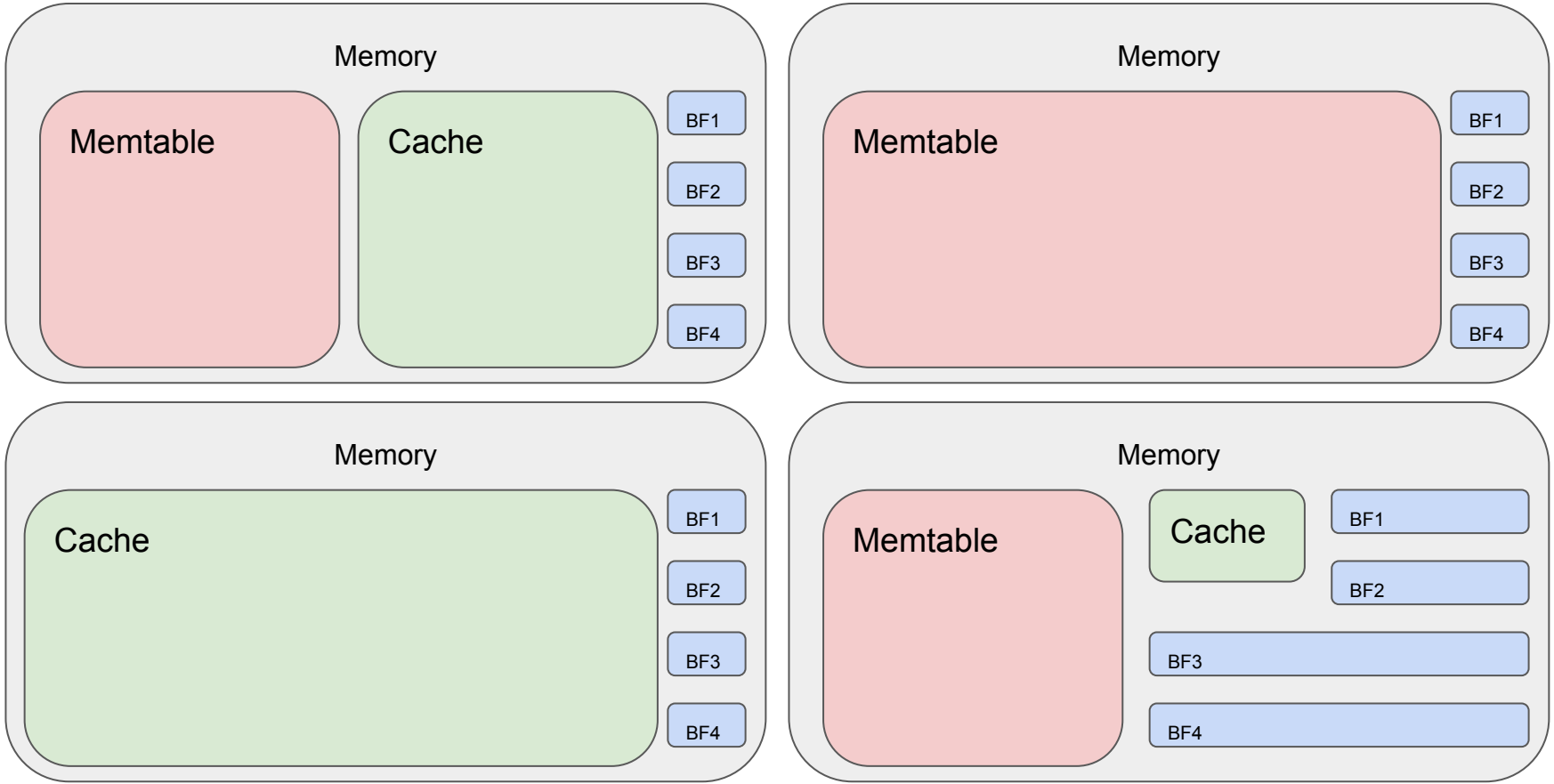
Optimal Memory Allocation

Mali Akmanalp, Sophie Hilgard, Andrew Ross

Motivation


Memory
Memtable
Cache
BF1
BF2
BF3
BF4
Memory
Memtable
BF1
BF2
BF3
BF4
Memory
Cache
BF1
BF2
BF3
BF4
Memory
Memtable
Cache
BF1
BF2
BF3
BF4

What does it depend on

What does it depend on

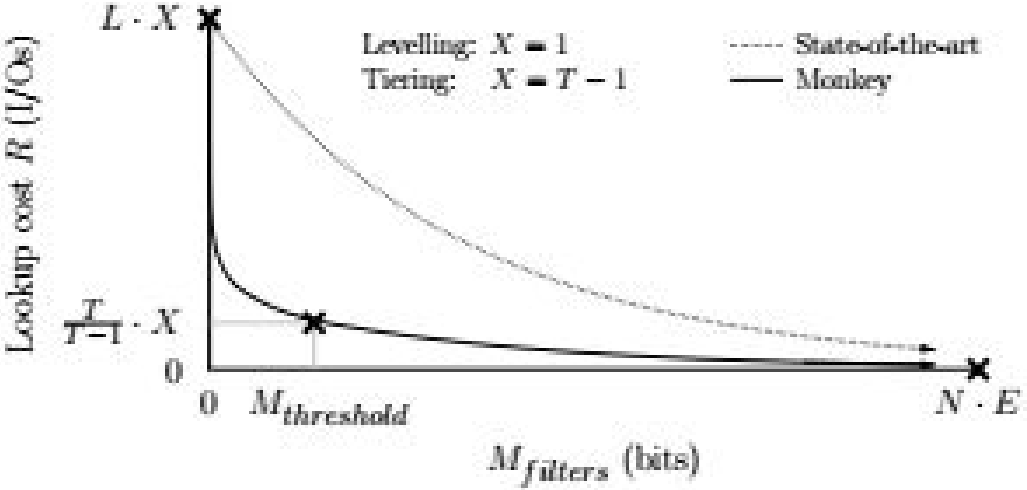
We know the right size matters:

Monkey: Optimal Navigable Key-Value Store

Niv Dayan
Harvard University
dayan@seas.harvard.edu

Manos Athanassoulis
Harvard University
manos@seas.harvard.edu

Stratos Idreos
Harvard University
stratos@seas.harvard.edu



But can we consider the LSM tree as a whole?

But can we consider the LSM tree as a whole?

But can we consider the LSM tree as a whole?

But can we consider the LSM tree as a whole?

But can we consider the LSM tree as a whole?

But can we consider the LSM tree as a whole?

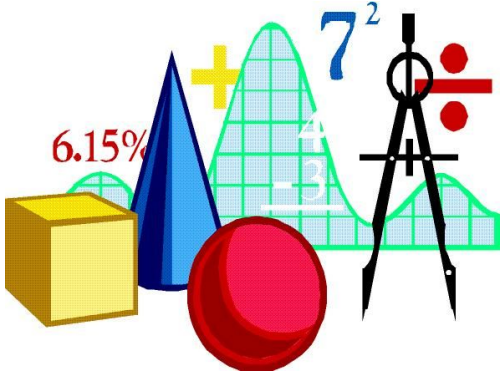
(+ the query distribution?)

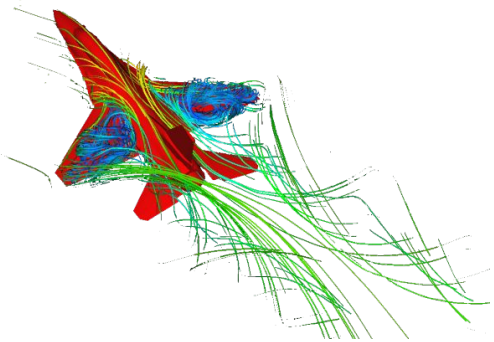
How much do optimal sizes depend on the query distribution?

How well do we need to know the query distribution?

High-level strategy

Modeling + Simulation + Benchmarks





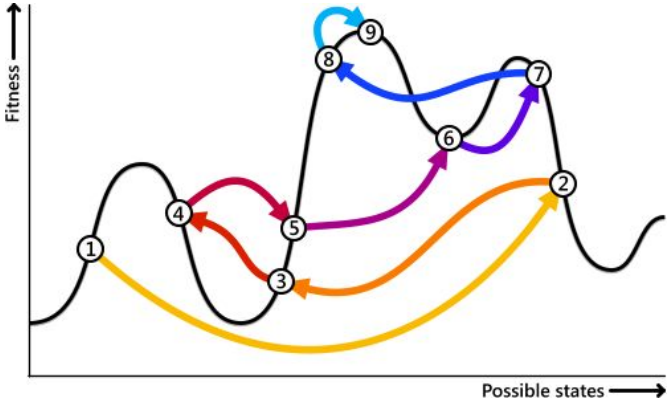
Make sure they all agree

High-level strategy

Then:

Learn what matters

Annealing + Inference?



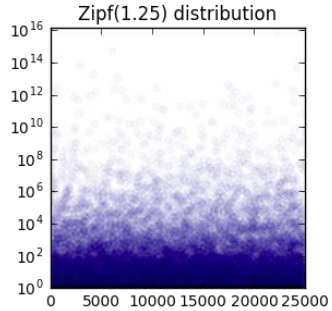
Implement in RocksDB

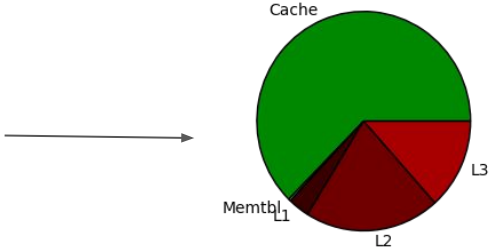
Preliminary Results

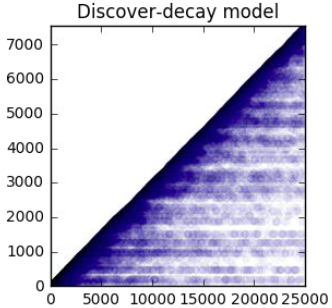
Simulation

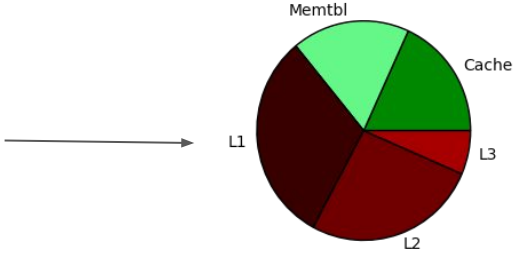
Simulated tree with cache, memtable, layers, and bloom filters w/ configurable sizes.

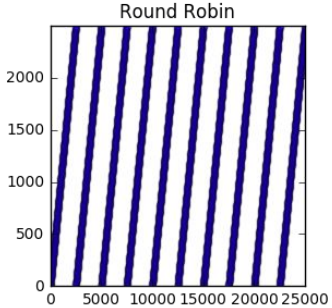
Same LSM tree architecture
+ Different query distributions
= Very different outcomes

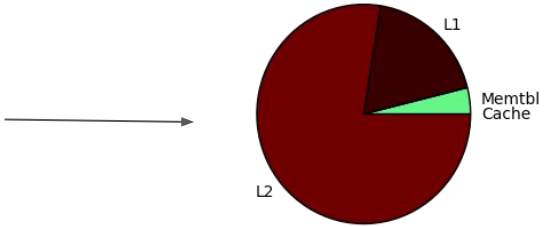






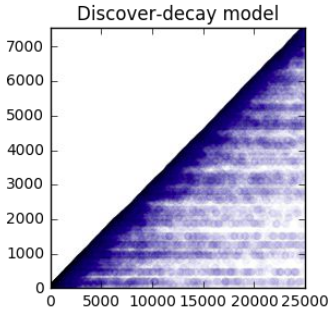




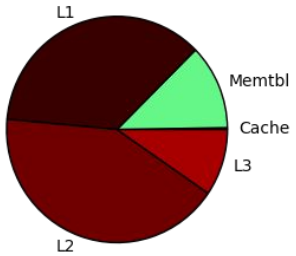


Simulation

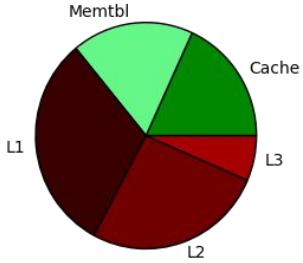
Different LSM tree architectures are best for different query distributions.



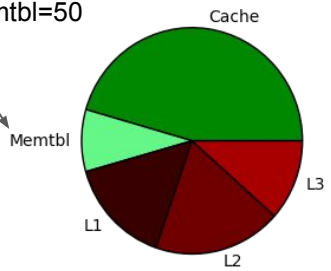
cache=0, memtbl=125



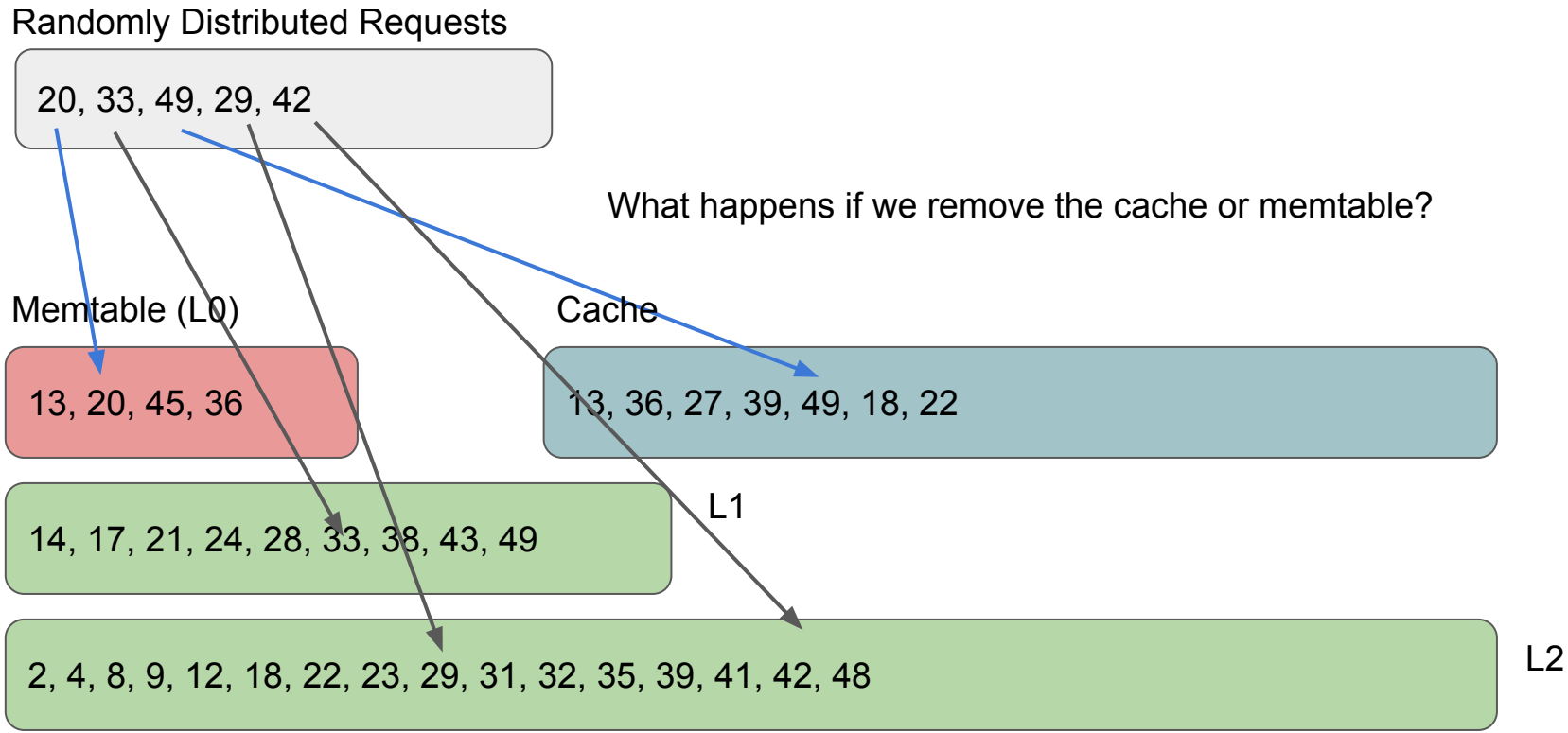
cache=25, memtbl=100



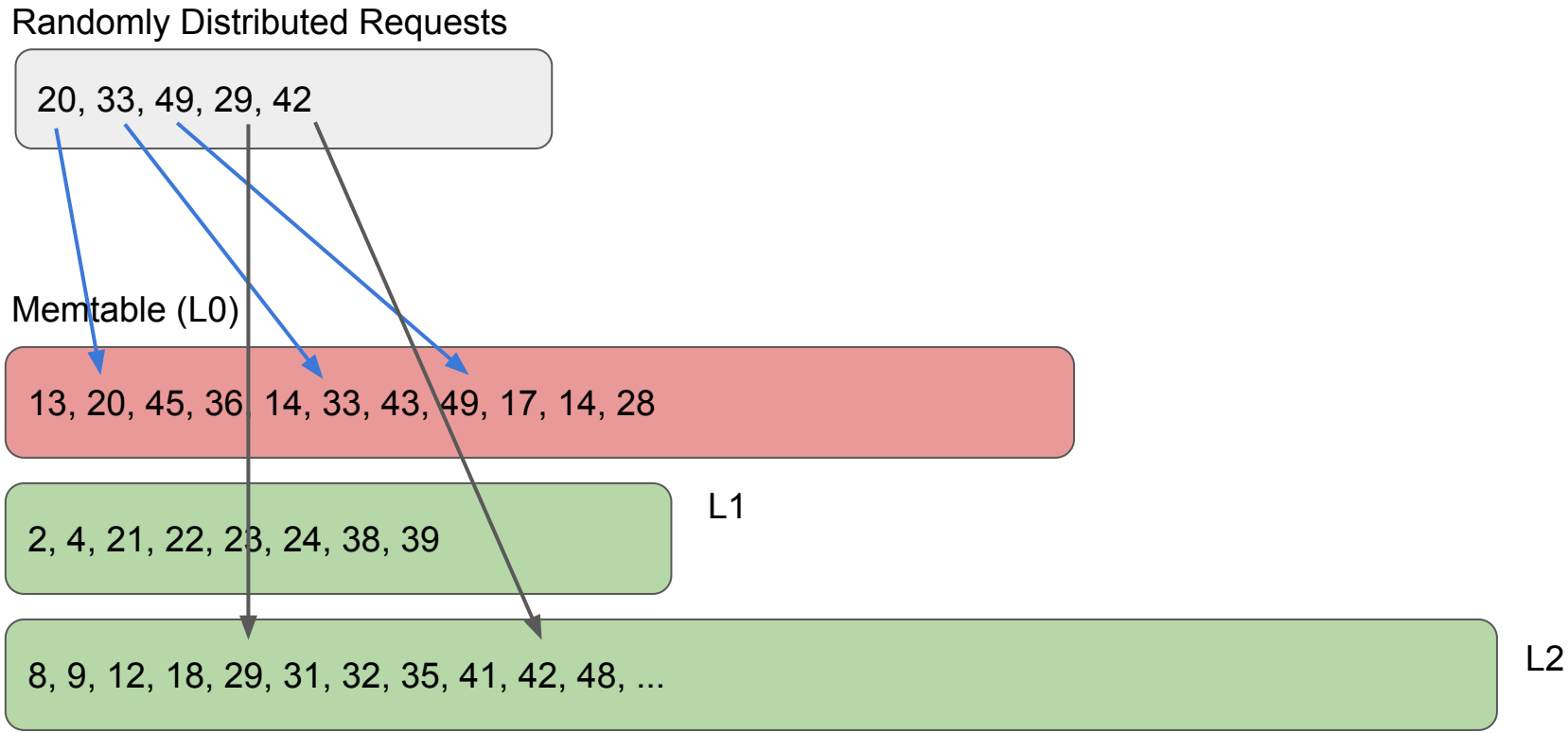
cache=75, memtbl=50



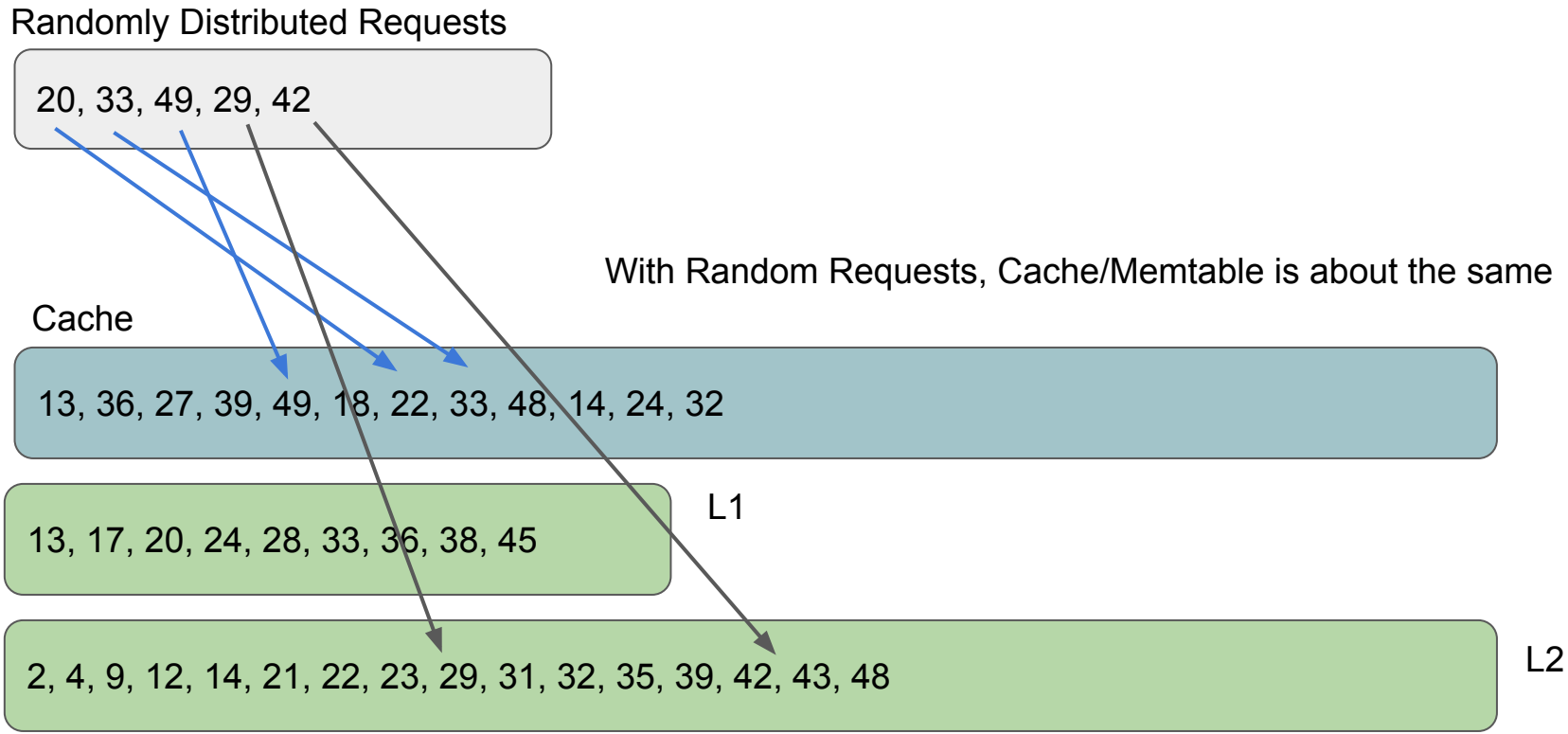
Modeling


Randomly Distributed Requests
20, 33, 49, 29, 42
What happens if we remove the cache or memtable?
Memtable (L0)
Cache
13, 20, 45, 36
13, 36, 27, 39, 49, 18, 22
14, 17, 21, 24, 28, 33, 38, 43, 49
L1
2, 4, 8, 9, 12, 18, 22, 23, 29, 31, 32, 35, 39, 41, 42, 48
L2

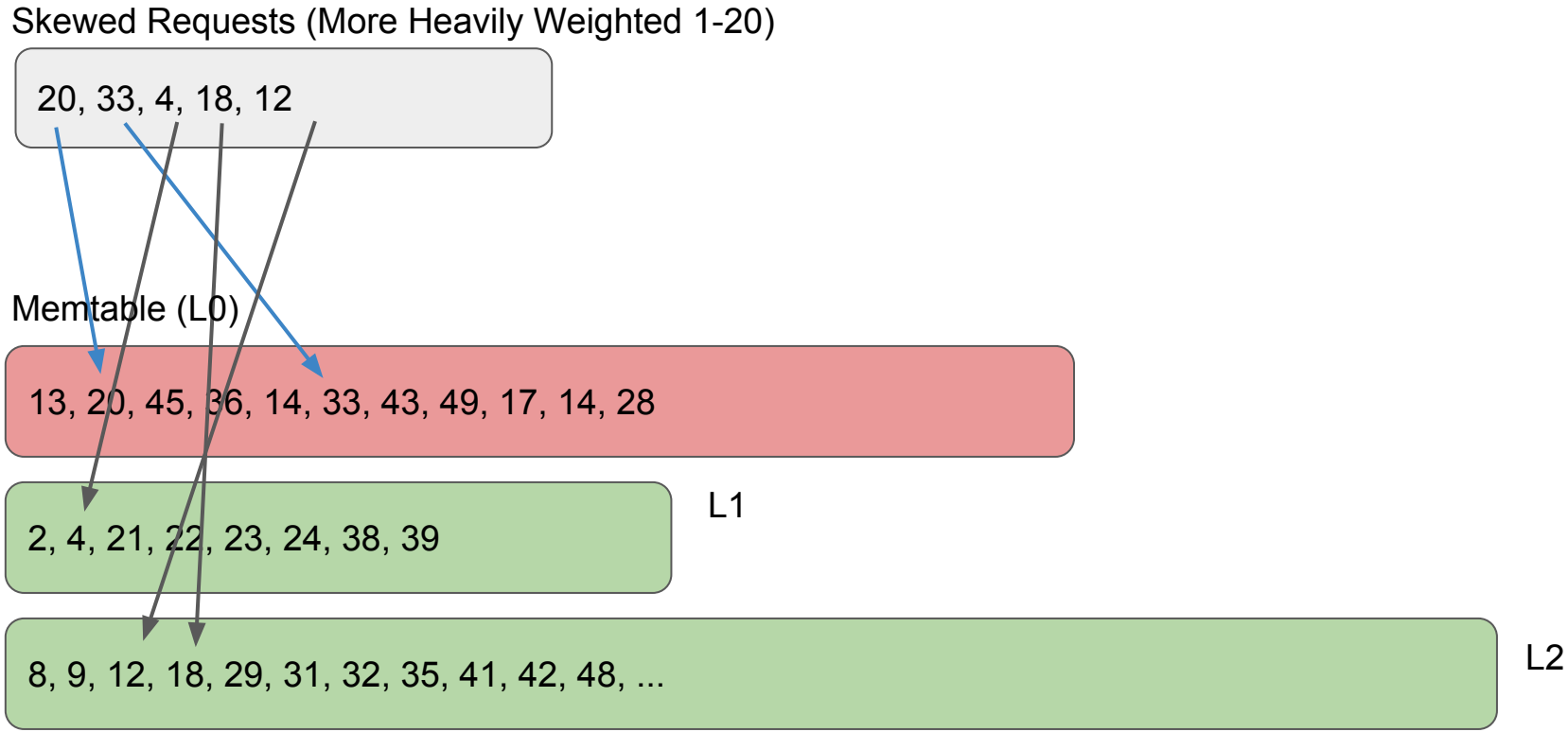
Modeling


Randomly Distributed Requests
20, 33, 49, 29, 42
Memtable (L0)
13, 20, 45, 36
14, 33, 43, 49, 17, 14, 28
2, 4, 21, 22, 23, 24, 38, 39
L1
8, 9, 12, 18, 29, 31, 32, 35, 41, 42, 48, ...
L2

Modeling


Randomly Distributed Requests
20, 33, 49, 29, 42
With Random Requests, Cache/Memtable is about the same
Cache
13, 36, 27, 39, 49, 18, 22, 33, 48, 14, 24, 32
13, 17, 20, 24, 28, 33, 36, 38, 45
L1
2, 4, 9, 12, 14, 21, 22, 23, 29, 31, 32, 35, 39, 42, 43, 48
L2

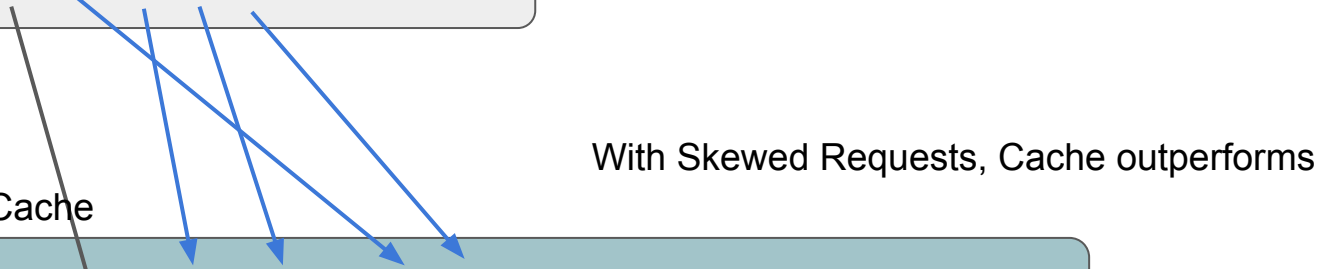
Modeling


Skewed Requests (More Heavily Weighted 1-20)
20, 33, 4, 18, 12
Memtable (L0)
13, 20, 45, 36, 14, 33, 43, 49, 17, 14, 28
2, 4, 21, 22, 23, 24, 38, 39
L1
8, 9, 12, 18, 29, 31, 32, 35, 41, 42, 48, ...
L2

Modeling

Skewed Requests (More Heavily Weighted 1-20)

20, 33, 4, 18, 12



With Skewed Requests, Cache outperforms

Cache

13, 17, 27, 4, 9, 18, 22, 33, 12, 14, 24, 2

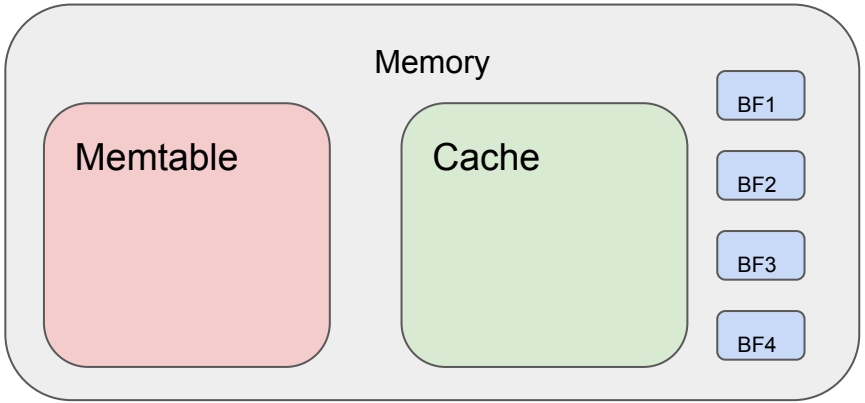
13, 17, 20, 24, 28, 33, 36, 38, 45

L1

2, 4, 9, 12, 14, 21, 22, 23, 29, 31, 32, 35, 39, 42, 43, 48

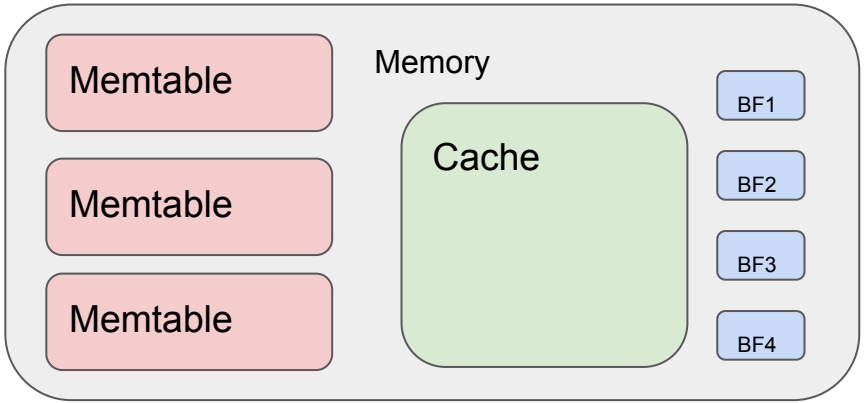
L2

Parameter space for experiment



- Memtable size
- Cache size
- Bloom filter size for each layer

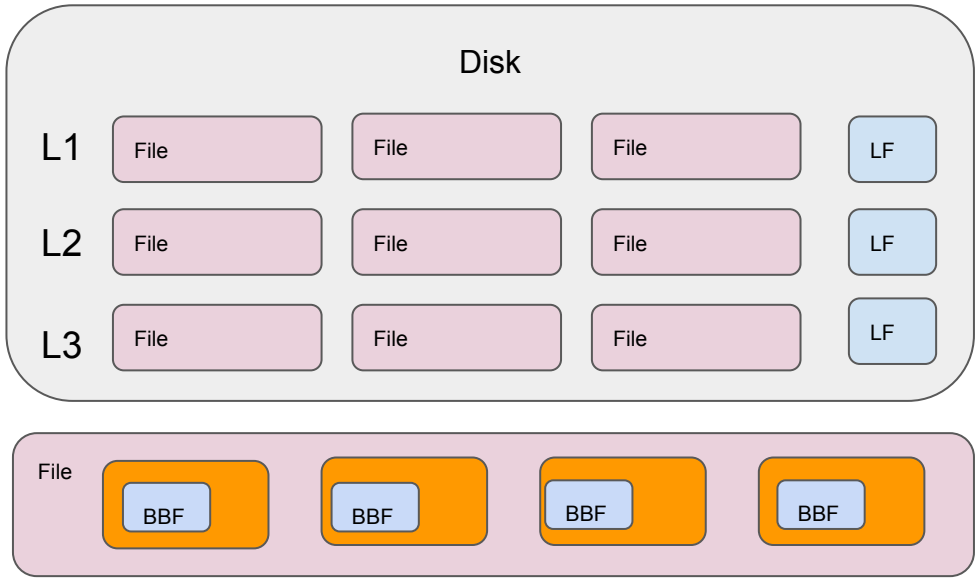
Memtables in RocksDB: “write buffers”



- **write_buffer_size**: memtable size
- **max_write_buffer_number:** number of memtables
- **min_write_buffer_number_to_merge** configure how often memtables get written

For us, memtable size is the total of all the write buffers.
We probably want the Vector memtable.

Bloom Filters in RocksDB



- **Block based bloom filter**
  For each block
- **Layer bloom filter:**
  For a full layer

For us, bloom filter size is the total of all layer filters.

Caches in RocksDB

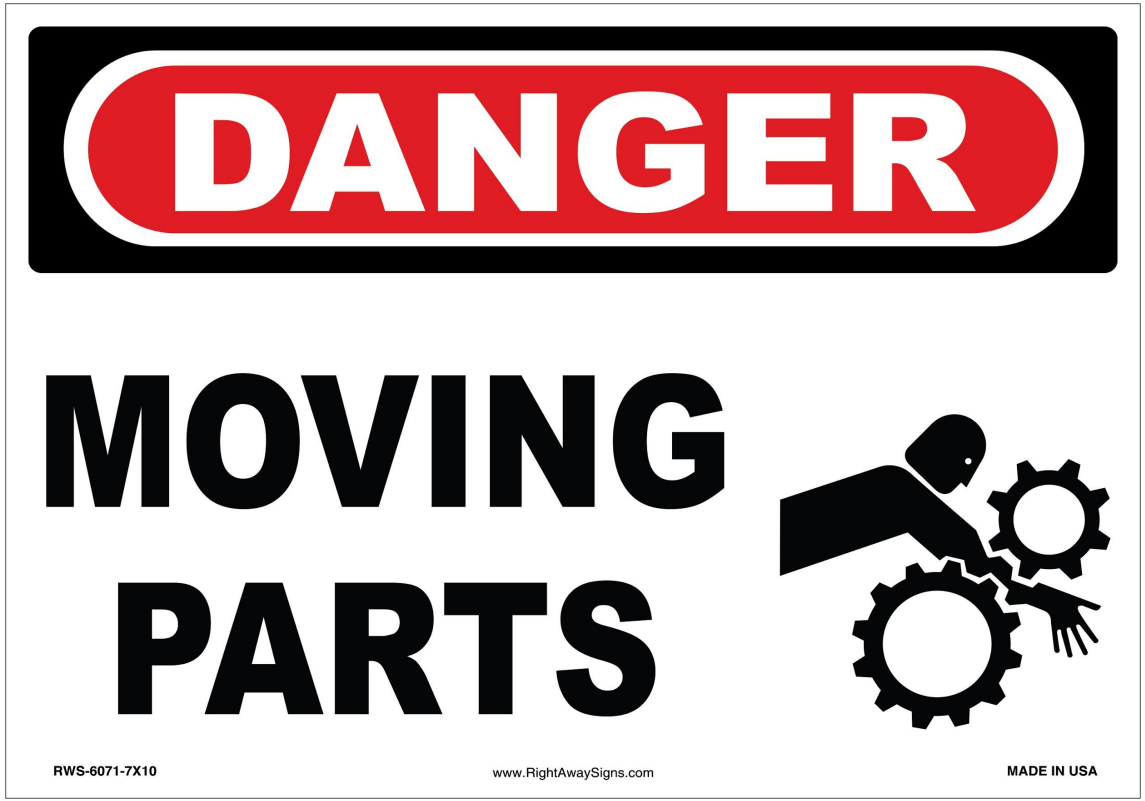
For us, cache size is the total block cache.

- **No key cache?**
- **Persistent read cache:** On-disk cache to look up blocks and files
- **Block cache:** Uncompressed blocks from files: LRU / Time
- **OS Page cache?** Unsure if we should try to bypass or include

Measurements

- Simulation Cache:
  - Neat feature that allows you to simulate cache behavior with different sizes on a real workload
- CreateDBStatistics()
  - Compaction Stats: Read/Write/Moved (GB)
  - General Stats: number / time spent of writes?
  - Block cache:
    - Hit/Miss rate
  - Layers:
    - Hit/miss for memtable
    - Hit for L1/L2/L2+ (interesting that it’s not for all layers)
  - Bloom Filter:
    - Bloom filter “usefulness”

Problems and Bottlenecks



MONKEY focused on changing just one thing.

How do we define success?

MONKEY VS. STATE-OF-THE-ART

FALSE POSITIVE RATES

	Level	State-Of-The-Art	Monkey
Buffer:	0	--	--
	1	0.82%	0.000028%
	2	0.82%	0.00028%
	3	0.82%	0.0028%
	4	0.82%	0.028%
	5	0.82%	0.28%
	6	0.82%	1.2%
Lookup cost:		*0.049 I/Os*	*0.015 I/Os*
Update cost:		0.105 I/Os	0.105 I/Os
Main memory:		80.002 GB	80.002 GB

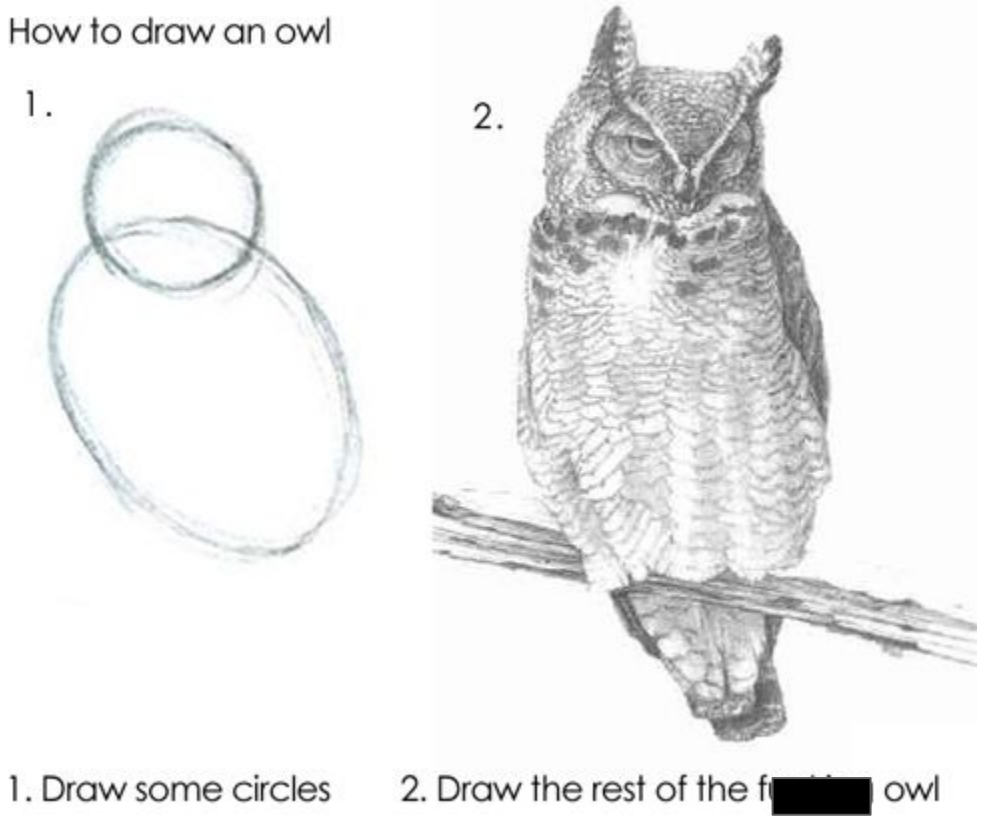
ENTRIES PER LEVEL

Monkey lookups are 3.32x faster!

Can we show similar results for different workloads and more memory options?

Preliminary Conclusions

Optimization is hard



But important

RocksDB offers good building blocks

but it might be hard to balance.

Actual quote: “Even we as RocksDB developers don't fully understand the effect of each configuration change”

Math + simulation + experimentation seems tractable.

Math + simulation + experimentation seems tractable.

hopefully.